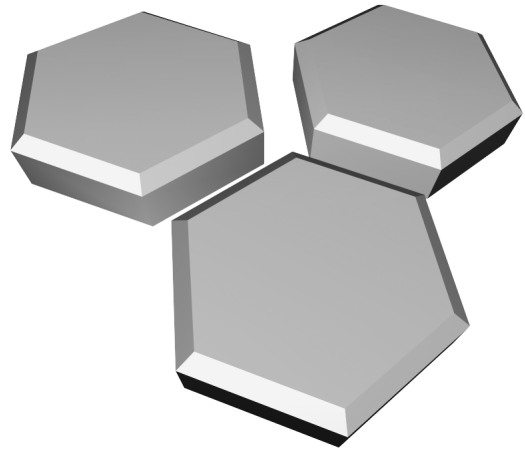
Das C++ Codebook



PEARSON Education

ADDISON-WESLEY

André Willms

Das C++ Codebook

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Falls alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

10 9 8 7 6 5 4 3 2 1
05 04 03

ISBN 3-8273-2083-6

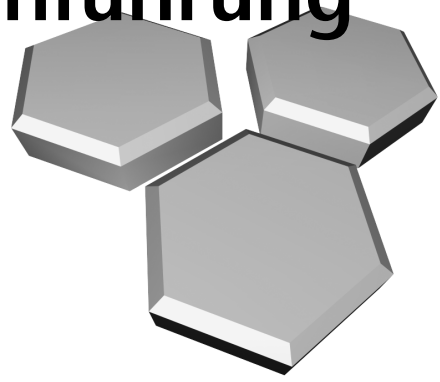
© 2003 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany
Alle Rechte vorbehalten

Korrektorat: Simone Meißner, Fürstenfeldbruck
Lektorat: Frank Eller, feller@pearson.de
Herstellung: Elisabeth Egger, eegger@pearson.de
Satz: reemers publishing services gmbh, Krefeld
Umschlaggestaltung: Marco Lindenbeck, mlindenbeck@webwo.de
Druck und Verarbeitung: Kösel, Kempten (www.KoeselBuch.de)

Printed in Germany

Inhaltsverzeichnis

Teil I
Einführung



Einführung

Sie interessieren sich für C++ im Allgemeinen und für dieses Codebook im Besonderen? Dann lesen Sie bitte die nächsten beiden Abschnitte, um besser entscheiden zu können, ob dieses Buch Ihren Vorstellungen entspricht.

An wen richtet sich dieses Buch?

Sie haben sich bereits mit C++ und der Objektorientierten Programmierung beschäftigt, sind in der Lage ein C++-Programm zu verstehen, aber es macht noch Schwierigkeiten, eigene Lösungsansätze zu finden und diese in C++ umzusetzen?

Sie können bereits eigene, auch größere, C++-Programme schreiben, haben dies auch schon getan, fragen sich aber, wie bestimmte alltägliche Probleme gelöst werden bzw. ob Ihr Lösungsansatz verbessert werden könnte?

Sie sind ein(e) alte(r) C++-Hase/-Häsin und wollten sich Anregungen für Problemlösungen holen, mit denen Sie sich bisher noch nicht befasst haben?

Dann werden Sie höchstwahrscheinlich in diesem Buch einige anregende Ansätze, Lösungsvorschläge und Implementierungsvarianten finden.

An wen richtet es sich nicht?

Sie haben gehört, dass C++ eine tolle Programmiersprache ist, sich aber bisher noch nicht mit dem Thema »Programmierung« befasst?

Sie können in C programmieren und wollen nun auf C++ umsatteln?

Sie besitzen eine langjährige Erfahrung als Programmierer(in), aber nicht in C++?

Sie haben schon einige große Projekte entwickelt, sind aber mit Objektorientierung nur marginal in Berührung gekommen?

Dann sollten Sie dieses Buch erst einmal weglegen, Ihr Geld in ein C++-Lehrbuch investieren und bei Bedarf anschließend hierauf zurückkommen.

Vielleicht finden Sie ja im Literaturverzeichnis am Ende dieses Kapitels einige Anregungen.

Was finden Sie in diesem Buch?

Eines vorweg: Dieses Buch ist kein Lehrbuch!

Wenn Sie noch kein grundlegendes Verständnis der Sprache C++ erlangt haben, dann wird Ihnen das vorliegende Buch dieses Grundverständnis mit sehr hoher Wahrscheinlichkeit auch nicht vermitteln können.

Dieses Buch ist eine Sammlung von Problemlösungen. Sie finden hier die Lösung für so einfache Probleme wie die Bestimmung, ob eine Zahl gerade ist oder wie die Fakultät einer Zahl berechnet wird.

Auch bei schwierigeren Themen, wie zum Beispiel ein String, der Groß- und Kleinschreibung nicht unterscheidet, oder auf welchen Tag Ostern fällt oder wie die Laufzeit eines Programms gemessen wird oder wie ein Personalausweis auf Gültigkeit geprüft wird, lässt Sie dieses Buch nicht allein.

Selbst für komplexe Themen wie die Implementierung einer Matrix oder eines höhenbalancierten Baums, der mit Iteratoren Inorder, Preorder, Postorder und Levelorder durchlaufen werden kann, oder der Lösung linearer Gleichungssysteme oder der Mustererkennung in Texten oder der Übertragung von Dateien von einem FTP- oder Web-Server finden Sie hier einen Lösungsansatz.

Für über 150 Probleme finden Sie in diesem Buch eine Lösung mit vollständig dokumentiertem, leicht nachvollziehbarem Quellcode, der mit zusätzlichen Erklärungen und kleinen Einführungen in die behandelte Thematik ergänzt ist.

Geschichte von C++

Ende der 60er Jahre entwickelte Martin Richards die Programmiersprache BCPL, die starken Einfluss auf die Anfang der 70er Jahre von Ken Thompson entworfene Sprache B hatte.

Obwohl B bereits für UNIX entwickelt wurde, stand sie Assembler noch recht nahe. Ihre prozessorabhängigen Sprachmerkmale und die damit verbundene Unfähigkeit der Portierung veranlassten Dennis Ritchie 1972, eine portierbare Sprache für UNIX zu entwickeln. Er nannte sie C.

Das UNIX-Projekt, an dem er arbeitete, wurde fast vollständig in C geschrieben.

Ende der 70er bis Anfang der 80er Jahre arbeitete Bjarne Stroustrup an einer Verbesserung der Sprache C: Sie sollte auch die Möglichkeit der Objektorientierung unterstützen. Es entstand »C mit Klassen« (C with classes.)

1983 setzt sich die Bezeichnung C++ durch. Der Name D wurde abgelehnt, weil er nicht die starke Beziehung zur Sprache C zum Ausdruck brachte.

1986 wird die erste Fassung von »Die C++-Programmiersprache« (The C++ Programming Language) von Bjarne Stroustrup veröffentlicht.

Die Standardisierungs-Bemühungen kamen langsam voran und gipfelten Ende 1997 in einer ersten ANSI-Norm, der 1999 eine Erweiterung folgte.

Kompatibilität

Bis auf die speziellen Themen Internet und Dateien, bei denen zwangsläufig auf plattformspezifische Funktionen zurückgegriffen werden musste, entsprechen alle Klassen und Funktionen der plattformunabhängigen ANSI-Norm.

Leider heißt das nicht automatisch, dass die Beispiele mit jedem Compiler problemlos zu kompilieren sind. Ein Beispiel:

```
for(int i=1; i<=10; ++i)
  cout << i << endl;

for(int i=1; i<=10; ++i)
  cout << i << endl;
```

Dieser Quellcode sollte – eingebettet in eine syntaktisch einwandfreie Funktion oder Methode – problemlos von einem die Norm beherrschenden Compiler übersetzt werden können.

Dummerweise meldet der Visual C++-Compiler noch in der Version 6 einen Fehler. Das liegt daran, dass der Compiler die Zählvariablen nicht als lokale Variablen des Schleifenblocks versteht, sondern fälschlicherweise als lokale Variablen des Blocks, in dem die Schleifen stehen. Deswegen bricht er mit der Behauptung ab, dass die Variable `i` neu definiert würde.

Eine andere Problematik zeigt der folgende Ausschnitt der Klasse `CShifter`:

```
class CShifter {
public:

  CShifter(const CShifter &s);

  template<typename CType>
  CShifter(const CType &c);
};
```

Auch hier ist von der Norm her klar, dass das Template nur dann zum Einsatz kommt, wenn der Datentyp des Konstruktor-Arguments nicht `CShifter` ist. Trotzdem bricht der Visual C++- Compiler in der Version 6 bei einem Gebrauch des ersten Konstruktors mit der Fehlermeldung »Mehrdeutiger Aufruf« ab.

Der im Buch und im Repository vorgestellte Quellcode wurde mit dem Visual C++.NET-Compiler fehlerfrei kompiliert.

Darüber hinaus finden Sie auf der CD alle im Buch vorgestellten Lösungen noch einmal in einer für den Visual C++ Version 6 Service Pack 5 kompilierbaren Form.

Sollte mit Ihrem Compiler keine der beiden Varianten fehlerfrei übersetzbar sein, so bleibt Ihnen leider keine andere Wahl, als eigenständig Veränderungen vorzunehmen.

Sollten die Quellcodes auf einem anderen Compiler nicht funktionieren oder Sie die Quellcodes für einen anderen Compiler lauffähig gemacht haben, würde ich mich über ein Feedback freuen.

Die CD zum Buch

Auf der CD finden Sie die Quellcodes aller im Buch aufgeführten Programme. Eine genaue Zuordnung, welche Funktion oder Klasse in welchen Dateien stehen, finden Sie bei dem jeweiligen Rezept.

Zusätzlich liegen die Quellcodes als herkömmliche .cpp- und .h-Dateien für den Visual C++.NET-Compiler im Verzeichnis /codes/completeNET-txt und für den Visual C++ 6 Service Pack 5 Compiler im Verzeichnis /codes/completeVS6-txt.

Literaturverzeichnis

Im Folgenden steht eine Aufführung einiger Bücher, die sowohl die Sprache C++ selbst als auch die im Buch behandelten Themen vertiefen.

DIN1355: Zeit – Kalender, Wochennumerierung, Tagesdatum, Uhrzeit

Engeln-Müllges, Gisela: Numerische Mathematik für Ingenieure / von Gisela Engeln-Müllges u. Fritz Reuter 5., überarb. Aufl. - Mannheim; Wien; Zürich: Bibliographisches Institut, 1987

Knuth, Donald E: The Art of Computer Programming – Volume 2 Seminumerical Algorithms, Third Edition, Addison-Wesley, 1998

Ottmann, Thomas: Algorithmen und Datenstrukturen / von Thomas Ottmann und Peter Widmayer, 2., vollst. überarb. und erw. Auflage, Mannheim; Leipzig; Wien; Zürich: BI-Wiss.-Verl., 1993

RFC0959: File Transfer Protocol (FTP)

RFC1123: Requirements for Internet Hosts – Application and Support

RFC1738: Uniform Resource Locators (URL)

RFC2616: Hypertext Transfer Protocol – HTTP/1.1

Sedgewick, Robert: Algorithmen in C++, München: Addison-Wesley, 1992

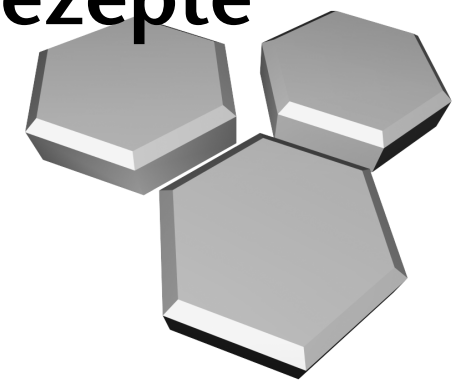
Stroustrup, Bjarne: Die C++-Programmiersprache, 4., überarb. und erw. Aufl., München: Addison-Wesley, 2000

Willms, André: C++-Programmierung, München: Addison-Wesley, 2001

Willms, André: C++ STL, Bonn: Galileo Press, 2000

Willms, André: Go To C++-Programmierung, München: Addison-Wesley, 1999

Teil II
Rezepte



Grundlagen

1 Wie wird ein C++-Programm gestartet?

Das Starten eines C++-Programms ist gleichbedeutend mit dem Aufruf der Funktion `main`. Ist die `main`-Funktion an ihrem Ende angelangt, ist auch das C++-Programm beendet. Daraus ergibt sich die logische Konsequenz, dass jedes C++-Programm eine `main`-Funktion besitzen muss. Die einfachste Form einer `main`-Funktion sieht wie folgt aus:

```
int main() {
}
```

Listing 1: Das einfachste C++-Programm

Vor dem Funktionsnamen steht der Rückgabewert der Funktion, in diesem Fall `int`. Hinter dem Funktionsnamen werden in runden Klammern eventuelle Funktionsparameter angegeben. Werden keine Funktionsparameter benötigt, bleibt das Klammernpaar leer.

Hinter dem Funktionskopf steht in geschweiften Klammern der Anweisungsblock der Funktion.

Die Funktion liefert keinen Wert zurück, obwohl hier ein `int`-Wert als Rückgabetyp definiert wurde. Dies ist nur bei der `main`-Funktion erlaubt, weil diese vom Compiler durch ein implizites `return` ergänzt wird. Ältere Compiler benötigen zum korrekten Beenden der Hauptfunktion unter Umständen noch ein explizites `return`.

```
int main() {
  return(0);
}
```

Listing 2: Eine Hauptfunktion mit explizitem return

2 Wie wird etwas ausgegeben?

Obwohl in C++ auch noch die in C übliche Ausgabe über `printf` zur Verfügung steht, wird die Ausgabe über das Objekt `cout` bevorzugt, welches in der Header-Datei `iostream` definiert ist. Dazu werden die auszugebenden Elemente mit dem Operator << in den Ausgabestrom »geschoben«.

```
std::cout << "Eine Ausgabe";
```

Dabei stehen Zeichenketten innerhalb doppelter Anführungszeichen.

Es können auch mehrere Ausgaben mit einer Anweisung getätigt werden.

```
std::cout << "Eine" << "Ausgabe";
```

Alle Elemente der C++-Standardbibliothek sind im Namensbereich `std` definiert. Um aber nicht immer den Namensbereich explizit angeben zu müssen, kann er mit der `using`-Anweisung global verfügbar gemacht werden:

```
using namespace std;
```

Das folgende Beispiel gibt den Text »Hello World« auf dem Bildschirm aus:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello World!";
}
```

Listing 3: Ausgabe des Textes »Hello World«

3 Wie werden Steuerzeichen ausgegeben?

Steuerzeichen werden in C++ in Form so genannter Escape-Sequenzen ausgegeben. Die folgende Tabelle gibt einen Überblick über die zur Verfügung stehenden Escape-Sequenzen.

Esc.Seq.	Zeichen
\a	**BEL** (bell), gibt ein akustisches Warnsignal
\b	**BS** (backspace), der Cursor geht eine Position nach links
\f	**FF** (formfeed), ein Seitenvorschub wird ausgelöst
\n	**NL** (new line), der Cursor geht zum Anfang der nächsten Zeile
\r	**CR** (carriage return), der Cursor geht zum Anfang der aktuellen Zeile
\t	**HT** (horizontal tab), der Cursor geht zur nächsten horizontalen Tabulatorposition
\v	**VT** (vertical tab), der Cursor geht zur nächsten vertikalen Tabulatorposition
\"	" wird ausgegeben
\'	' wird ausgegeben
\?	**?** wird ausgegeben
\\	\ wird ausgegeben

Tabelle 1: Die Steuerzeichen in C++

Das folgende Beispiel gibt den Text »Hello World« in zwei Zeilen aus:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hello\nWorld!\n";
}
```

Listing 4: Ausgabe von »Hello World« in zwei Zeilen

4 Welche Variablentypen gibt es?

C++ unterscheidet verschiedene Variablentypen, die im Folgenden aufgeführt sind.

Ganzzahlige Typen

Die ganzzahligen Typen können – wie der Name bereits zum Ausdruck bringt – nur ganzzahlige Werte, also Werte ohne Nachkommastellen, aufnehmen. Es existieren die Datentypen `short`, `int` und `long`. Der Compiler muss garantieren, dass eine `short`-Variable mindestens 16 Bit besitzt. Ansonsten gilt nur die Relation `short<=int<=long`.

Jeder dieser drei Datentypen existiert als vorzeichenbehafteter (`signed`) oder vorzeichenloser (`unsigned`) Wert. Wird weder `signed` noch `unsigned` explizit angegeben, so ist der Datentyp vorzeichenbehaftet.

Um einen ganzzahligen Wert explizit als `long`-Wert zu kennzeichnen, kann das Suffix `l` oder `L` angegeben werden:

```
long x=234L;
```

Fließkommatypen

Auch bei den Fließkommatypen stehen drei Möglichkeiten zur Verfügung; `float`, `double` und `long double`. Die Größen der Variablen stehen in der Relation `float<=double<=long double`. Fließkommatypen sind immer vorzeichenbehaftet.

Bei konkreten Werten können `float`-Werte durch das Suffix `f` oder `F` und `long double`-Werte durch das Suffix `l` oder `L` gekennzeichnet werden.

```
float x=3.14F;
double y=3.14;
long double z=3.24L;
```

Boolesche Typen

Als boolescher Typ steht der Datentyp `bool` zur Verfügung. Ein boolescher Typ kann nur die Werte `true` oder `false` aufnehmen. Dabei steht `true` intern für den Wert 1 und `false` für den Wert 0.

```
bool b=true;
```

Zeichentypen

Ein Zeichen kann mit dem Datentyp `char` gespeichert werden. Zeichenliterale stehen in einfachen Anführungszeichen. `char` besitzt eine minimale Größe von 8 Bit. Ohne explizite Angabe ist der Datentyp char vorzeichenlos. Eine vorzeichenbehaftete Variante steht mit `signed char` zur Verfügung.

```
char c='A';
```

Mit Hilfe des `sizeof`-Operators lässt sich die Größe eines Datentyps bestimmen.

```
#include <iostream>

using namespace std;

int main() {
  cout << "char        : " << sizeof(char) << endl;
  cout << "bool        : " << sizeof(bool) << endl;
  cout << "short       : " << sizeof(short) << endl;
  cout << "int         : " << sizeof(int) << endl;
  cout << "long        : " << sizeof(long) << endl;
  cout << "float       : " << sizeof(float) << endl;
  cout << "double      : " << sizeof(double) << endl;
  cout << "long double : " << sizeof(long double) << endl;
}
```

Listing 5: Auflisten der Größe aller Datentypen

5 Wie werden Kommentare eingefügt?

In C++ gibt es zwei Möglichkeiten, Kommentare zu definieren: die einzeiligen Kommentare mit `//` und die mehrzeiligen Kommentare mit `/* ... */`

```
/*
main-Funktion, die nichts macht
*/
int main() {
```

Listing 6: Der Einsatz von Kommentaren

```
return(0);   // Explizites Return
}
```

Listing 6: Der Einsatz von Kommentaren (Forts.)

Die mehrzeiligen Kommentare dürfen nicht verschachtelt werden.

6 Welche Kontrollstrukturen gibt es?

Im Folgenden werden die Kontrollstrukturen von C++ aufgeführt.

if ... else

Die `if`-Anweisung wird eingesetzt, um eine Verzweigung im Programm zu erzeugen. Dabei wird der Programmfluss von einer Bedingung abhängig gemacht.

```
if(bedingung) {
// Anweisungsblock 1
}
else {
// Anweisungsblock 2
}
```

Sollte die hinter `if` formulierte Bedingung wahr sein, so wird der Anweisungsblock 1 ausgeführt. Andernfalls wird Anweisungsblock 2 abgearbeitet. Danach geht der Programmfluss hinter dem `if`-Konstrukt weiter. Abbildung 1 zeigt den Sachverhalt.

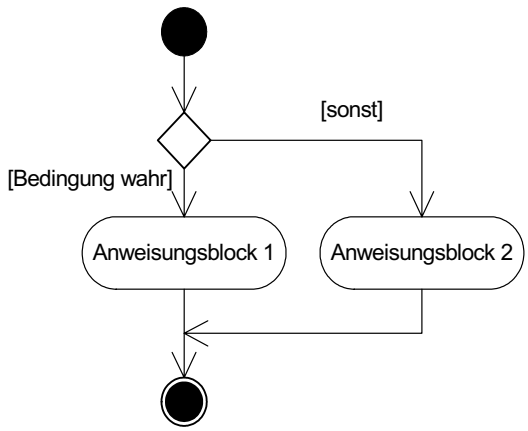


Abbildung 1: Die if-Anweisung

Der `else`-Block ist optional. Wenn er jedoch benutzt wird, dann muss er immer einem `if`-Block folgen.

Zur Formulierung einer Bedingung stehen diverse Vergleichsoperatoren zur Verfügung, die in folgender Tabelle zusammengefasst sind.

Operator	Bedeutung
<	kleiner
<=	kleiner gleich
==	gleich
!=	ungleich
>=	größer gleich
>	größer
!	Negation

Tabelle 2: Die Vergleichsoperatoren

Einzelne Bedingungen können mit logischen Operatoren zu einer Gesamtbedingung verknüpft werden. Die beiden in C++ verfügbaren Operatoren sind in der unteren Tabelle aufgeführt.

Operator	Bedeutung
||	logisches inklusives Oder
&&	logisches Und

Tabelle 3: Die logischen Operatoren

Das folgende Beispiel demonstriert die Anwendungsweise:

```
if ( (x!=0) && (1.0/x > 0.00001) )
{ /*...*/ }
```

Die logischen Operatoren besitzen eine so genannte Kurzschlusseigenschaft. Als Konsequenz sind die logischen Operatoren in C++ nicht kommutativ. `a&&b` ist also nicht gleich `b&&a`.

Im Detail lässt sich die Kurzschlusseigenschaft am letzten Beispiel betrachten. Sollte der Wert von `x` gleich Null sein, so wäre die erste Bedingung falsch. Weil eine Und-Bedingung aber nur dann wahr wird, wenn jede Einzelbedingung wahr ist, kann die Gesamtbedingung mit `x` gleich Null nicht mehr wahr werden. Es macht daher programmtechnisch keinen Sinn, die zweite Bedingung noch auf ihren Wahrheitsgehalt zu prüfen. Und dass die zweite Bedingung in diesem Fall tatsächlich nicht geprüft wird, bezeichnet man als Kurzschlusseigenschaft.

Somit wird die verbotene Division durch Null in der zweiten Bedingung mit `x` gleich Null nicht ausgeführt. Ohne Kurzschlusseigenschaft wäre dies nicht gewährleistet.

while

`while` ist ein Schleifenkonstrukt, mit dem Wiederholungen aufgebaut werden können:

```
while(bedingung) {
// Anweisungsblock
}
```

Abbildung 2 zeigt den Programmfluss von `while`.

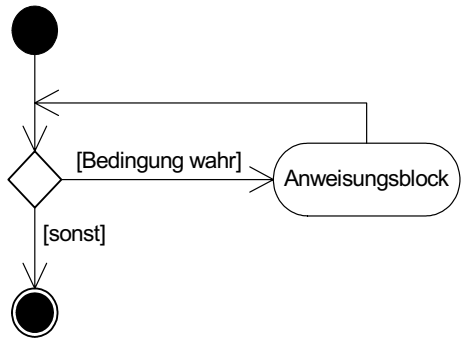


Abbildung 2: Die while-Anweisung

Beim Betreten der `while`-Schleife wird zunächst die Bedingung geprüft. Sollte sie wahr sein, dann wird der Anweisungsblock von `while` ausgeführt und anschließend die Bedingung erneut geprüft. Ist die Bedingung irgendwann einmal falsch, bricht die `while`-Schleife ab. Das folgende Beispiel gibt die Zahlen von eins bis zehn auf dem Bildschirm aus.

```
int x=1;
while(x<=10) {
  cout << x << endl;
  x++;
}
```

Für alle Anweisungsblöcke – mit Ausnahme der Funktionsanweisungsblöcke – gilt die Vereinfachung, dass auf die geschweiften Klammern verzichtet werden kann, wenn der Anweisungsblock nur aus einer Anweisung besteht:

```
int x=1;
while(x<=10)
  cout << x++ << endl;
```

Die `while`-Schleife existiert noch in einer Sonderform, der `do-while`-Schleife:

```
do {
// Anweisungsblock
} while (bedingung);
```

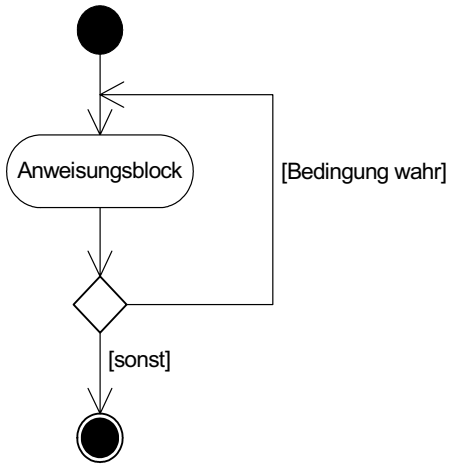


Abbildung 3: Die do-while-Anweisung

Im Gegensatz zur `while`-Schleife wird zuerst der Anweisungsblock betreten und dann die Bedingung geprüft. Damit wird der Anweisungsblock immer mindestens einmal ausgeführt, unabhängig vom Wahrheitsgehalt der Bedingung. Abbildung 3 stellt den Programmfluss dar.

for

Die komplexeste Schleifenvariante in C++ ist die `for`-Schleife. Ihr abstrakter Aufbau sieht wie folgt aus:

```
for(initialisierung; bedingung; iteration) {
// Anweisungsblock
}
```

Abbildung 4 stellt die `for`-Schleife im Programmfluss dar.

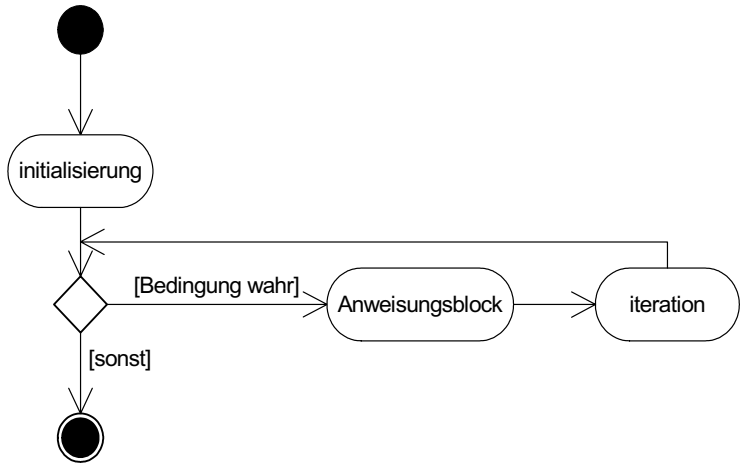


Abbildung 4: Die for-Schleife

Das Beispiel einer von eins bis zehn zählenden Schleife kann mit einer `for`-Schleife wie folgt realisiert werden:

```
for(int x=1; x<=10; x++)
  cout << x << endl;
```

Das Beispiel macht von der Vereinfachung Gebrauch, dass die geschweiften Klammern bei nur einer Anweisung im Anweisungsblock weggelassen werden können.

switch

Mit der `switch`-Anweisung kann auf verschiedene Werte einer Variablen reagiert werden:

```
switch(x) {
  case 5: cout << "Der Wert ist 5" << endl;
          break;
  case 23:  cout << "Der Wert ist 23" << endl;
            break;
  case 99:  cout << "Der Wert ist 99" << endl;
            break;
  default:  cout << "Einer der anderen Werte" << endl;
            break;
}
```

Sollte `x` einen Wert haben, der durch kein `case` abgedeckt ist, dann werden die Anweisungen hinter `default` ausgeführt. Die Angabe von `default` in einem `switch`-Block ist optional.

Bei `case` und `default` handelt es sich um Sprungmarken. Wären sie nicht mit einem `break` abgeschlossen, so würden alle Anweisungen hinter dem angesprungenen `case` oder `default` ausgeführt.

7 Wie werden Arrays definiert?

Arrays werden mit dem Operator [ ] definiert. Innerhalb der eckigen Klammern wird definiert, wie viele Elemente das Array besitzen soll.

```
int f[20];
```

In C++ besitzt das erste Element eines Feldes immer den Index 0. Die Elemente des oben definierten 20 Elemente großen Feldes haben damit die Indizes 0 bis 19.

Nachdem ein Array definiert wurde, kann auf die einzelnen Elemente mit dem Indexoperator zugegriffen werden:

```
f[5]=11;
```

Hier wird dem sechsten Element der Wert 11 zugewiesen.

Mehrdimensionale Arrays werden durch die Angabe mehrerer Dimensionen definiert:

```
int d[10][5][2];
```

Das letzte Beispiel definiert ein dreidimensionales Feld mit insgesamt 100 Elementen.

8 Wie werden Verweistypen definiert?

Im Gegensatz zu den Wertetypen, die einen Wert tatsächlich speichern, sind die Verweistypen lediglich eine Referenz auf eine Entität, die wieder ein Werte- oder Verweistyp sein kann.

Zeiger

Eine Form von Verweistyp sind die in C++ Zeiger genannten Typen. Sie werden bei der Definition mit einem `*` definiert:

```
int *p;
```

Die Variable p ist nun ein Zeiger und kann damit die Adresse eines Datentyps speichern. In C++ besitzen alle Variablen einen Datentyp, weswegen bei einem Zeiger angegeben werden muss, von welchem Datentyp er die Adressen speichern darf. Im Falle des Zeigers `p` ist dies der Variablentyp `int`.

Die Adresse einer Variablen wird – mit Ausnahme von Arrays, deren Name allein bereits für ihre Adresse steht – mit dem Adressoperator `&` ermittelt.

```
int x;
p=&x;
```

Der Zeiger `p` hat nun die Adresse von `x` gespeichert und verweist damit auf `x`. Mit dem Dereferenzierungsoperator `*` kann jetzt über den Zeiger `p` der Wert von `x` ausgelesen und verändert werden:

```
*p=55;
cout << *p << endl;
```

Im Normalfall werden Zeiger im Zusammenhang mit der dynamischen Speicherverwaltung eingesetzt. Der Operator `new` liefert die Adresse eines dynamisch angelegten Objektes, die dann in einem Zeiger gespeichert wird:

```
int *ptr = new int;
```

Über die Dereferenzierung kann dann die dynamisch angelegte `int`-Variable angesprochen werden. Freigegeben wird ein dynamisch angelegter Speicherbereich mit `delete`:

```
delete ptr;
```

Referenzen

Mit den Referenzen hat C++ im Vergleich zu C eine Erweiterung erfahren. Im Gegensatz zu Zeigern, die die Adresse einer Variablen speichern und im Laufe ihres Lebens auch unterschiedliche Adressen aufnehmen können, sind Referenzen fest an ein Objekt gebunden. Man spricht auch von einem Alias:

```
int x;
int &r=x;
r=20;
cout << x;
```

Das obere Beispiel gibt den Wert 20 aus. Interessant ist die vereinfachte Syntax der Referenzen. Bei der Herstellung des Verweises wird keine Adresse mehr ermittelt und beim Zugriff über die Referenz keine Dereferenzierung mehr benötigt.

9 Wie werden zusammengesetzte Datentypen definiert?

Zusammengesetzte Datentypen heißen in C++ Strukturen. Sie werden mit dem Schlüsselwort `struct` definiert:

```
struct Person {
  char vorname[40];
  char nachname[40];
  int alter;
};
```

Besonderes Augenmerk gilt dem Semikolon hinter der schließenden geschweiften Klammer.

Durch das Definieren einer Struktur wurde gewissermaßen ein neuer Datentyp definiert, von dem eine Variable angelegt werden kann:

```
Person p;
```

Auf die einzelnen Elemente der Person `p` wird mit dem Element-Operator `.` (auch Member-Operator genannt) zugegriffen:

```
p.alter=22;
```

Wird eine Struktur-Variable einer anderen Struktur-Variablen zugewiesen, dann wird eine flache Kopie angefertigt:

```
Person p2=p;
```

Auch ein Zeiger auf Strukturen kann definiert werden und ermöglicht damit das dynamische Anlegen von Struktur-Variablen:

```
Person *pz = new Person;
```

Auf die Elemente der dynamisch angelegten Struktur-Variablen wird durch Dereferenzierung zugegriffen:

```
(*pz).alter=22;
```

Die Klammerung ist notwendig, weil der Element-Operator stärker bindet als der Dereferenzierungsoperator. Wegen der komplizierten Syntax wurde der Operator -> eingeführt:

```
pz->alter=22;
```

10 Welche String-Arten stehen in C++ zur Verfügung?

In C++ stehen grundsätzlich zwei Arten von Strings zur Verfügung: die von C übernommenen C-Strings und die in C++ definierten Strings.

C-Strings

C-Strings werden in Arrays vom Datentyp `char` gespeichert:

```
char n[80];
```

Da Arrays nicht zugwiesen werden können, existiert eine Funktion `strcpy`, mit deren Hilfe ein C-String kopiert werden kann. Um sie nutzen zu können, muss die Headerdatei `cstring` eingebunden werden:

```
strcpy(n,"Andre Willms");
cout << n << endl;
```

Eine mit doppelten Anführungszeichen definierte Zeichenkette ist automatisch ein C-String.

Die Besonderheit eines C-Strings liegt in seiner Ende-Kennung. Die Ende-Kennung besitzt den Wert 0 und steht hinter dem letzten Zeichen des C-Strings. Ein C-String hat seine Länge also nicht explizit gespeichert. Es müssen die Zeichen bis zur Endekennung gezählt werden, um die Länge eines C-Strings zu ermitteln. Und genau dies erledigt die Funktion `strlen`:

```
cout << strlen(n) << endl;
```

Viele Funktionen der Standardbibliothek arbeiten mit C-Strings, sodass man hin und wieder mit ihnen zu tun hat.

Strings

In C++ wurde eine spezielle Klasse `string` definiert. Ihre Deklaration steht in der Headerdatei `string`. Einem String kann problemlos ein C-String zugewiesen werden:

```
string s="Andre";
cout << s << endl;
```

Strings besitzen die Methode `c_str`, die den String als C-String zurückliefert:

```
strcpy(n,s.c_str());
cout << n << endl;
```

Weil Strings und C-Strings beliebig ineinander konvertiert werden können, werden die folgenden Funktionen, soweit es Sinn macht, ausschließlich mit Strings arbeiten.

11 Wie kann ein String eingelesen werden?

Für Strings ist der >>-Operator nicht überladen. In Anlehnung an `cin.getline` werden wir uns eine Funktion `getString` mit folgenden Parametern schreiben:

- `s` ist der String, in den hineingeschrieben wird.
- `a` ist die maximale Anzahl an Zeichen, die eingelesen werden.

Der Quellcode ist im Folgenden aufgelistet.

```
void getString(string &s, int a) {

/*
** Puffer für die Aufnahme der Zeichen anlegen
*/
  char *buff=new char[a];

/*
** Mit der getline-Methode von cin in
** den Puffer lesen
*/
```

Listing 7: Die Funktion getString

```
  cin.getline(buff,a);

/*
** Text in String kopieren und String freigeben
*/
  s=buff;
  delete(buff);
}
```

Listing 7: Die Funktion getString (Forts.)

Die Funktion `getString` basiert auf der Eingabe über `cin.getline`. Anhand der maximalen Zeichenanzahl wird dynamisch ein Puffer angelegt, der von `getline` beschrieben wird. Der gefüllte Puffer wird danach dem String zugewiesen. Zum Schluss wird der reservierte Puffer wieder freigegeben.

`getstring` befindet sich auf der CD in den `StringFunctions`-Dateien.

12 Wie wird ein String in Großbuchstaben umgewandelt?

Um einen String in Großbuchstaben umzuwandeln wird von der Funktion `toupper` aus der Headerdatei `cctype` Gebrauch gemacht. Die Funktion `toupper` liefert den entsprechenden Großbuchstaben zurück, wenn das übergebene Zeichen ein Kleinbuchstabe ist. Sollte es sich nicht um einen Kleinbuchstaben handeln, dann wird das übergebene Zeichen unverändert zurückgeliefert.

```
string toUpperString(const string &org) {
  string s=org;
  size_t length=s.size();
  string::iterator iter=s.begin();
  while(length--) {
    *iter=toupper(*iter);
    ++iter;
  }
  return(s);
}
```

Listing 8: Die Funktion toUpperString

Die Funktion `toUpperString` macht vom übergebenen String eine Kopie, durchläuft diese und ruft für jedes Zeichen `toupper` auf. Nachdem die Kopie komplett in Großbuchstaben umgewandelt wurde, wird sie von der Funktion zurückgegeben.

Eine Funktion `toLowerString`, die einen String in Kleinbuchstaben umwandelt, ist nun kein Problem mehr. Es wird lediglich die Funktion `toupper` durch `tolower` ersetzt:

```
string toLowerString(const string &org) {
  string s=org;
  size_t length=s.size();
  string::iterator iter=s.begin();
  while(length--) {
    *iter=tolower(*iter);
    ++iter;
  }
  return(s);
}
```

Listing 9: Die Funktion toLowerString

Die beiden Funktionen finden Sie auf der CD in den `StringFunctions`-Dateien.

Das folgende Beispiel setzt die Funktionen ein:

```
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {

  string s;
  cout << "Eingabe:";
  getString(s,100);
  cout << "\nAusgabe toUpperString(): "
       << toUpperString(s) << endl;
  cout << "Ausgabe toLowerString(): "
       << toLowerString(s) << endl;
}
```

Listing 10: Ein Beispiel zu toUpperString und toLowerString

Die `main`-Funktion finden Sie auf der CD unter Buchdaten\Beispiele\main0203.cpp. Sie macht von `getString` aus Rezept 11 Gebrauch. Ein Ausgabe-Beispiel zeigt Abbildung 5.

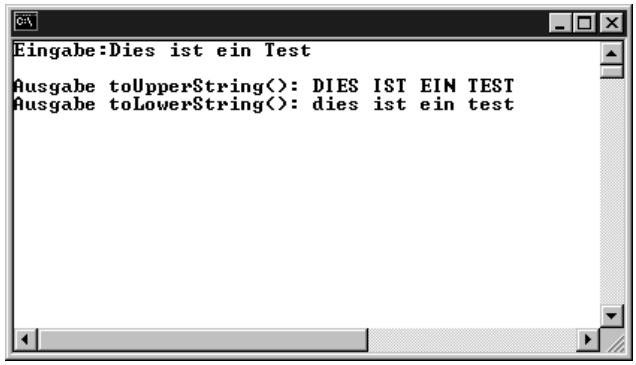


Abbildung 5: Die Ausgabe von toUpperString und toLowerString

13 Wie wird der erste Buchstabe jedes Worts eines Satzes in einen Großbuchstaben umgewandelt?

Statt eines kompletten Strings kann auch der erste Buchstabe eines jeden Worts im String in einen Großbuchstaben umgewandelt werden. Das würde dann folgendermaßen aussehen: »In Diesem Satz Ist Der Erste Buchstabe Immer Groß.«

Eine Funktion `capitalize`, die einen übergebenen String in Kapitälchen umwandelt, könnte wie folgt aussehen:

```
string capitalize(const string &org) {
  string s=org;
  size_t length=s.size();

/*
** Erstes Zeichen auf Großbuchstaben setzen
*/
  if(length)
    s[0]=toupper(s[0]);
```

Listing 11: Die Funktion capitalize

```
/*
** Alle Zeichen des Strings durchlaufen
*/
  for(size_t i=1;i<length; ++i) {

/*
** Aktuelles Zeichen ein Buchstabe und
** das vorherige Zeichen kein Buchstabe?
**  => Aktuelles Zeichen in Großbuchstaben umwandeln
*/
    if(isalpha(s[i])&&!isalpha(s[i-1]))
      s[i]=toupper(s[i]);
    else
      s[i]=tolower(s[i]);
  }
  return(s);
}
```

Listing 11: Die Funktion capitalize (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

Sollte der String aus mehr als einem Zeichen bestehen, dann wird das erste Zeichen grundsätzlich in einen Großbuchstaben umgewandelt.

Interessant ist auch die Bedingung, die darüber entscheidet, ob in einen Groß- oder Kleinbuchstaben umgewandelt wird:

```
if(isalpha(s[i])&&!isalpha(s[i-1]))
```

Sollte das aktuelle Zeichen ein Buchstabe sein, das Zeichen davor aber nicht, dann wird das aktuelle Zeichen in einen Großbuchstaben umgewandelt. Andernfalls wird daraus ein Kleinbuchstabe gemacht.

Wegen dieser Bedingung werden auch Wörter, die hinter einem Bindestrich stehen, mit einem Großbuchstaben begonnen. Wollte man nur Leerzeichen als gültigen Trenner erlauben, dann müsste die Bedingung `!isalpha` in `isspace` umgewandelt werden.

Ein Beispiel könnte wie folgt aussehen:

```
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {

  string s;
  cout << "Eingabe:";
  getString(s,100);
  cout << "\nAusgabe capitalize(): "
       << capitalize(s) << endl;
}
```

Listing 12: Ein Beispiel zu capitalize

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0204.cpp*. Sie macht von `getString` aus Rezept 11 Gebrauch.

14 Wie wird ein String in einzelne Wörter zerlegt?

Für diese Aufgabe schreiben wir eine Funktion `chopIntoWords`, die einen Vektor mit den einzelnen Wörtern zurückliefert. Die `vector`-Klasse ist in der Headerdatei `vector` definiert. Der STL-Algorithmus `find_first_of` muss über die Headerdatei `algorithm` zugänglich gemacht werden.

Der zweite Parameter `separators` ist ein String, der alle gültigen Wort-Trenner enthält. Als Standard-Wert besitzt er `" \t\n"`.

```
vector<string> chopIntoWords(const string &org,
                             const string &separators) {

  vector<string> words;
  string::const_iterator en, be=org.begin();

  do {
```

Listing 13: Die Funktion chopIntoWords

```
/*
** Nach dem ersten Separator-Zeichen suchen
*/
    en=find_first_of(be,org.end(),
                     separators.begin(),separators.end());

/*
** Steht ein Wort vor dem gefundenen Separator?
*/
    if(be!=en) {

/*
** String zu Vektor hinzufügen und Wort hineinkopieren
*/
      words.push_back("");
      copy(be,en,back_inserter(words.back()));
    }

/*
** Ab dem gefundenen Separator-Zeichen alle direkt
** nachfolgenden Separator-Zeichen überspringen
*/
    be=find_first_not_of(en,org.end(),
                         separators.begin(),separators.end());
  } while(be!=org.end());

  return(words);
}
```

Listing 13: Die Funktion chopIntoWords (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien. Sie macht von `find_first_not_of` aus Rezept 33 Gebrauch.

Der statische String `separators` definiert die Trennzeichen, die in Frage kommen. In diesem Fall wurden nur das Leerzeichen, der Tabulator und das Newline-Zeichen berücksichtigt. Sollen beispielsweise auch mit Bindestrich verbundene Wörter als einzelne Wörter behandelt werden, dann braucht der Bindestrich nur zu diesen Zeichen hinzugefügt werden. Der String `separators` wurde als statisch deklariert, weil er bei jedem Funktionsaufruf denselben Inhalt hat und daher nicht immer wieder erneut angelegt werden muss. Eine andere Variante wäre, ihn als zusätzlichen Funktionsparameter zu definieren, um eine größere Flexibilität zu erhalten.

Die Iteratoren `be` und `en` definieren den Beginn und das Ende eines erkannten Wortes. Zu Beginn wird `be` auf das erste Zeichen des Strings gesetzt.

In der Schleife wird dann im String nach dem ersten Zeichen gesucht, das einem Trennzeichen entspricht. Das so ermittelte Wort wird in den Vektor kopiert und der `be`-Iterator von `en` aus auf das erste Zeichen gesetzt, das kein Trennzeichen ist.

Auf diese Weise verarbeitet `chopIntoWords` auch Strings mit mehreren Trennzeichen zwischen zwei Wörtern korrekt.

Eine passende `main`-Funktion könnte so aussehen:

```
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {

  cout << "Eingabe:";
  string s;
  getString(s,100);
  cout << endl;
  vector<string> v=chopIntoWords(s);
  for(vector<string>::iterator i=v.begin(); i!=v.end(); ++i)
    cout << "\"" << *i << "\"" << endl;
}
```

Listing 14: Eine main-Funktion für chopIntoWords

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0205.cpp*. Sie macht von `getString` aus Rezept 11 Gebrauch.

Kann bei der Ausgabe auf die Anführungszeichen um die separierten Wörter herum verzichtet werden, dann lässt sich die Ausgabe eleganter mit einem Ostream-Iterator umsetzen:

```
copy(v.begin(), v.end(), ostream_iterator<string>(cout,"\n"));
```

Ein Ausgabe-Beispiel zeigt Abbildung 6

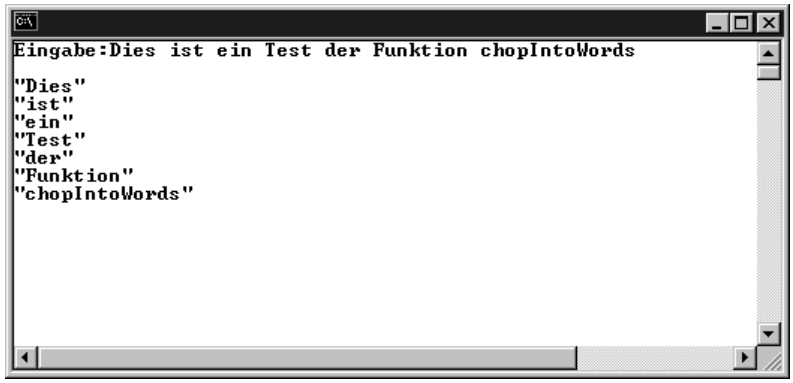


Abbildung 6: Die Ausgabe von chopIntoWords

15 Wie wird eine Zahl in einen String umgewandelt (1)?

Befassen wir uns zunächst mit der einfacheren Frage, wie eine numerische Zahl, zum Beispiel 55, in den String `"55"` umgewandelt werden kann. Dazu wird die Funktion `toString` implementiert, die dazu einen String-Stream einsetzt. Um String-Streams einsetzen zu können, muss die Headerdatei `sstring` eingebunden werden.

```
string toString(long val) {
  ostringstream o;
  o << val;
  return(o.str());
}
```

Listing 15: Die Funktion toString

Ein Objekt vom Typ `ostringstream` verhält sich wie ein normaler Ausgabestrom, nur dass die Ausgabe in einen Zeichenpuffer geschrieben wird. Mit Hilfe seiner Methode `str` wird der Zeichenpuffer als String zurückgegeben.

Auf diese Weise lässt sich jeder Datentyp, der die Möglichkeit der normalen Konsolenausgabe über `cout` bietet, in einen String umwandeln. Es bietet sich daher an, für diese Aufgabe direkt ein Template zu schreiben:

```
template<class Type>
string toString(Type val) {
  ostringstream o;
  o << val;
  return(o.str());
}
```

Listing 16: Das Template toString

Um auch die Möglichkeit zu besitzen, die Ausgabe mit einer festen Breite, einer Ausrichtung und einem Füllzeichen zu versehen, wird folgendes Template hinzugefügt:

```
template<class Type>
std::string toString(Type val,
                     int width,
                     bool left=false,
                     char fill=' ') {
  std::ostringstream o;
  if(left)
    o << setfill(fill) << setw(width) << std::left << val;
  else
    o << setfill(fill) << setw(width) << val;
  return(o.str());
}
```

Listing 17: Das Template toString mit Breiten- und Füllzeichenangabe

Die Templates finden Sie auf der CD in der *StringFunctions.h*-Datei.

16 Wie wird eine Zahl in einen String umgewandelt (2)?

In dieser zweiten Variante soll das Problem behandelt werden, wie eine Zahl, z.B. 143, in ihre ausgeschriebene Form, also »Einhundertdreiundvierzig«, umgewandelt werden kann.

Diese auf den ersten Blick aufwändige Problematik lässt sich schnell in den Griff bekommen, wenn wir uns die Regelmäßigkeiten der ausgeschriebenen Zahlen genauer ansehen.

Die Zahlen von 1 bis 12 haben keine Gemeinsamkeiten. Die Zahlen von 13 bis 19 besitzen jedoch – mit Ausnahme der 17 – die Regelmäßigkeit einer+»zehn«. Nur die 17 fällt aus dem Rahmen, denn sie heißt ja »Siebzehn« und nicht »Siebenzehn«.

Mit diesen Informationen lässt sich eine Funktion `upTo19` schreiben, die alle Zahlen bis 19 abdeckt:

```
string upTo19(unsigned long val, bool one=true) {
  switch(val) {
    case 1: if(one)
              return("eins");
            else
              return("ein");

    case 2: return("zwei");
    case 3: return("drei");
    case 4: return("vier");
    case 5: return("fuenf");
    case 6: return("sechs");
    case 7: return("sieben");
    case 8: return("acht");
    case 9: return("neun");
    case 10: return("zehn");
    case 11: return("elf");
    case 12: return("zwoelf");
    case 17: return("siebzehn");
    default: return(upTo19(val%10)+"zehn");
  }
}
```

Listing 18: Die Funktion upTo19

Da die Zahl 1 für sich alleine »Eins« heißt, im Zusammenhang mit anderen Zahlen (z.B. 21) aber »Ein«, wurde in `upTo19` eine boolesche Variable als zusätzlicher Funktionsparameter angelegt, der diesen Unterschied kontrolliert.

Die Zahlen von 20 bis 99 lassen sich in das Format einer+»und«+zehner pressen. Es sei denn, es handelt sich um glatte Zehnerwerte, dann wird nur der Zehner ausgegeben.

Diesen Umstand setzt die Funktion `upTo99` um:

```
string upTo99(unsigned long val, bool one) {

  if(val<20)
      return(upTo19(val,one));

  string word;
  if(val%10)
    word=upTo19(val%10,false)+"und";

  switch(val/10) {
    case 2: word+="zwanzig"; break;
    case 3: word+="dreissig"; break;
    case 4: word+="vierzig"; break;
    case 5: word+="fuenfzig"; break;
    case 6: word+="sechzig"; break;
    case 7: word+="siebzig"; break;
    case 8: word+="achtzig"; break;
    case 9: word+="neunzig"; break;
  }
  return(word);
}
```

Listing 19: Die Funktion upTo99

Abgesehen von glatten Hunderter-Werten gehorchen die Zahlen von 100 bis 999 der Regel upTo19+»hundert »+upTo99. Die Funktion `upTo999` bringt dies zum Ausdruck:

```
string upTo999(unsigned long val, bool one) {
  if(val<100)
    return(upTo99(val,one));

  string word=upTo19(val/100,false);
  word+="hundert";

  if(val%100)
    word+=upTo99(val%100, one);

  return(word);
}
```

Listing 20: Die Funktion upTo999

Die Zahlen von 1000 bis 999999 lassen sich durch upTo999+»tausend«+upTo999 erzeugen. Natürlich müssen glatte Tausender-Werte speziell berücksichtigt werden:

```
string underMillion(unsigned long val) {
  string word;
  if(val>=1000) {
    word=upTo999(val/1000, false);
    word+="tausend";
    }

  if(val%1000)
    word+=upTo999(val%1000, true);

  return(word);
}
```

Listing 21: Die Funktion underMillion

Der nächste Schritt sind Zahlen von 1.000.000 bis 999.999.999. Hier muss darauf geachtet werden, dass es im Singular »Million«, im Plural aber »Millionen« heißt. Die Funktion `underBillion` setzt dies um:

```
string underBillion(unsigned long val) {
  string word;
  if(val>=1000000) {
      unsigned long  mil=val/1000000;
      if(mil==1)
        word="einemillion";
      else {
        word=upTo999(mil, false);
        word+="millionen";
        }
      val%=1000000;
    }
  word+=underMillion(val);
  return(word);
}
```

Listing 22: Die Funktion underBillion

Der nächste Schritt ist nun nur noch ein Klacks. Auch hier muss auf den Unterschied zwischen »Milliarde« und »Milliarden« geachtet werden. Die Funktion `underTrillion` erledigt das:

```
string underTrillion(unsigned long val) {
  string word;
  if(val>=1000000000) {
    unsigned long mil=val/1000000000;
    if(mil==1)
      word="einemilliarde";
    else {
      word=upTo999(mil,false);
      word+="milliarden";
      }
    val%=1000000000;
  }
  word+=underBillion(val);
  return(word);
}
```

Listing 23: Die Funktion underTrillion

Wir haben nun den gesamten Wertebereich einer `unsigned long`-Variablen abgedeckt. Es fehlt nur noch eine Funktion, die die entsprechenden Aufrufe tätigt und die Null berücksichtigt. Nennen wir sie `toWord`:

```
string toWord(unsigned long val, bool upcase=true) {
  string word;
  if(val==0)
    word=("null");
  else
    word=underTrillion(val);

  if(upcase)
    word[0]=toupper(word[0]);

  return(word);
}
```

Listing 24: Die Funktion toWord

In `toWord` wird zusätzlich mit der booleschen Variable `upcase` die Möglichkeit geschaffen, die umgewandelten Zahlen mit einem Groß- oder einem Kleinbuchstaben beginnen zu lassen.

Die Funktionen finden Sie auf der CD in den *StringFunctions*-Dateien.

Das folgende Beispiel gibt die Zweier-Potenzen von 0-20 aus:

```
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  unsigned long x=1;
  for(int i=0; i<=20; i++, x*=2)
    cout << x << " heisst " << toWord(x) << endl;
}
```

Listing 25: Ein Beispiel zu toWord

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0207.cpp.*

Das Ergebnis ist in Abbildung 7 zu sehen.

Abbildung 7: Ein Beispiel zu toWord

17 Wie implementiert man einen String, der nicht zwischen Groß- und Kleinschreibung unterscheidet?

Um einen nicht sensitiven String zu erzeugen, gibt es mehrere Möglichkeiten. Die aufwändigste Variante wäre mit Sicherheit das Programmieren einer komplett neuen String-Klasse.

Um einen eleganteren Ansatz zu finden, schauen wir uns einmal die Definition des Strings aus der Standardbibliothek an:

```
typedef basic_string<char,
                     char_traits<char>,
                     allocator<char> > string;
```

Listing 26: Die Definition eines Strings in der Standardbibliothek

Ein String ist demnach nichts anderes als eine aus dem Template `basic_string` generierte Klasse.

Der erste Parameter des Templates definiert den Datentyp eines einzelnen Zeichens.

Der zweite Parameter bestimmt die `char_traits`, die Eigenschaften des Zeichen-Typs.

Der dritte spezifiziert den einzusetzenden Allokator. Der Allokator bildet die Schnittstelle zwischen einem STL-Container und der Speicherverwaltung.

Wenn wir das `char_traits`-Template genauer unter die Lupe nehmen, dann finden wir dort in den Methoden `eq` und `lt` die Definitionen, wie zwei Zeichen verglichen werden sollen:

```
static bool eq(const char& c1, const char& c2) {
  return (c1 == c2);
}

static bool lt(const char& c1, const char& c2) {
  return (c1 < c2);
}
```

Listing 27: Die Funktionen eq und lt bei char_traits<char>

Wenn wir anstelle eines Vergleichs `c1==c2` einfach `tolower(c1)==tolower(c2)` schreiben würden, dann wäre die Groß- und Kleinschreibung beim Vergleichen egal.

Wir schreiben daher eine eigene Klasse `char_traits`, die unsere Funktionalität beinhaltet. Die `char_traits` stellen darüber hinaus noch weitere Methoden zur Verfügung, die unsere eigene Klasse auch besitzen muss.

```
struct insensitive_char_traits {
  typedef char char_type;
  typedef int int_type;
};
```

Listing 28: Die eigene Klasse char_traits

Schauen wir uns den Aufbau der in `char_traits` enthaltenen Methoden im Detail an. Die Methode `assign` weist einer Zeichen-Variablen den Wert einer anderen zu:

```
  static void assign(char& c1, const char& c2) {
    c1 = c2;
  }
```

Listing 29: Die neue Methode assign

Die bereits vorhin vorgestellten Methoden `eq` und `lt` vergleichen zwei Zeichen. Dabei prüft `eq` (Abk. für equal) auf Gleichheit und `lt` (Abk. für less than) führt einen Vergleich auf kleiner durch:

```
  static bool eq(const char& c1, const char& c2) {
    return (tolower(c1) == tolower(c2));
  }

  static bool lt(const char& c1, const char& c2) {
    return (tolower(c1) < tolower(c2));
  }
```

Listing 30: Die neuen Methoden eq und lt

Die Methode `compare` vergleicht eine bestimmte Anzahl von Zeichen unter Zuhilfenahme der Methoden `eq` und `lt`:

```
static int compare(const char *s1, const char *s2, size_t len) {
  for (; 0 < len; --len, ++s1, ++s2)
    if (!eq(*s1, *s2))
      return (lt(*s1, *s2) ? -1 : +1);
  return (0);
}
```

Listing 31: Die neue Methode compare

Mittels `length` lässt sich die Anzahl der Zeichen in einem String ermitteln. Anhand der Abbruchbedingung der Schleife ist zu erkennen, dass die Zeichen innerhalb der String-Klasse wie C-Strings verwaltet werden. Dazu wird `eq` benötigt:

```
static size_t length(const char *s) {
  size_t len;
  for (len = 0; !eq(*s, char()); ++s)
    ++len;
  return (len);
}
```

Listing 32: Die neue Methode length

Über `copy` lassen sich mehrere Zeichen kopieren. Der Kopiervorgang selbst wird auf `assign` zurückgeführt:

```
static char *copy(char *s1, const char *s2, size_t len) {
  char *ptr = s1;
  for (; 0 < len; --len, ++ptr, ++s2)
    assign(*ptr, *s2);
  return (s1);
}
```

Listing 33: Die neue Methode copy

Mit der Methode `find` wird nach einem Zeichen in einem String gesucht. Der Vergleich wird mit `eq` durchgeführt:

```
static const char *find(const char *s,
                        size_t len,
                        const char& c) {
  for (; 0 < len; --len, ++s)
    if (eq(*s, c))
      return (s);
  return (0);
}
```

Listing 34: Die neue Methode find

`move` macht eigentlich nichts anderes als `copy`, nur mit der zusätzlichen Eigenschaft, dass sich bei `move` Quell- und Zielbereich überlappen dürfen. Für den Kopiervorgang wird wieder `assign` verwendet:

```
static char *move(char *s1, const char *s2, size_t len) {
  char *ptr = s1;
  if (s2 < ptr && ptr < s2 + len)
    for (ptr += len, s2 += len; 0 < len; --len)
      assign(*--ptr, *--s2);
  else
    for (; 0 < len; --len, ++ptr, ++s2)
      assign(*ptr, *s2);
  return (s1);
}
```

Listing 35: Die neue Methode move

Die folgende Variante von `assign` füllt einen Bereich mit einem Zeichen:

```
static char *assign(char *s, size_t len, char c) {
  char *ptr = s;
  for (; 0 < len; --len, ++ptr)
    assign(*ptr, c);
  return (s);
}
```

Listing 36: Die neue Methode assign für mehrere Zeichen

Die folgenden Funktionen wandeln zwischen dem Zeichen- und dem Metatyp. Sie sind für uns nicht weiter von Bedeutung:

```
static char to_char_type(const int_type& i) {
  return (i);
}

static int_type to_int_type(const char& c) {
  return (c);
}

static bool eq_int_type(const int_type& c1,const int_type& c2) {
  return (c1 == c2);
}

static int_type  eof() {
  return ((int_type)EOF);
}

static int_type not_eof(const int_type& i) {
  return (i != eof() ? i : !eof());
}
```

Listing 37: Methoden zum Umwandeln von Zeichen- und Meta-Typ

Es wäre auch möglich gewesen, unsere neue Klasse von `char_traits<char>` abzuleiten. Dann hätten wir uns die Implementierung einiger Funktionen erspart. Wir wären aber nicht daran vorbeigekommen, die Methoden `eq` und `lt` sowie alle Methoden, die von `eq` und `lt` Gebrauch machen, neu zu schreiben. Insofern war die komplett neue Implementierung nicht viel mehr Arbeit und gibt dabei tiefere Einblicke in die Funktionsweise der `char_traits`.

Um nun tatsächlich einen nicht sensitiven String einsetzen zu können, müssen wir unseren neuen Typ zuerst mit einem `typedef` deklarieren und unsere eigenen `char_traits` einsetzen:

```
typedef basic_string<char,
                     insensitive_char_traits,
                     allocator<char> > insensitive_string;
```

Listing 38: Die Klasse insensitive_string

Den `insensitive_string` finden Sie auf der CD in der Datei *Istring.h*.

Zum Abschluss noch ein kleiner Test unseres Strings:

```
#include "IString.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  insensitive_string s1="Test";
  insensitive_string s2="tEsT";
  if(s1==s2)
    cout << "Klappt!" << endl;
}
```

Listing 39: Ein Test von insensitive_string

Es braucht nicht erwähnt zu werden, dass der Text »Klappt!« tatsächlich ausgegeben wird. Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0208.cpp*.

Interessant ist auch, wie sich der nicht-sensitive String in Kombination mit dem Standard-String verhält:

```
insensitive_string s1="Test";
string s2="tEsT";
if(s1==s2)
  cout << "Strings sind gleich!" << endl;
```

Listing 40: Probleme bei der Kombination von string und insensitive_string

Bei der Kompilierung stellt sich heraus, dass sie – zumindest auf diese Weise – zunächst einmal nicht kombinierbar sind. Um exakt die oben verwendete Schreibweise einsetzen zu können, müssen sich zwangsläufig die Vergleichsoperatoren überladen.

Ein anderer Ansatz ist das Umwandeln einer der beiden Strings in einen C-String mit `c_str`. Diese Vorgehensweise hat den Vorteil, dass bei einer Kombination von sensitiven und nicht-sensitiven Strings gewählt werden kann, ob Groß- und Klein-

schreibung berücksichtigt werden soll oder nicht, je nachdem, welcher String in einen C-String umgewandelt wird:

```
if(s1==s2.c_str())
  cout << "Strings sind gleich!" << endl;

if(s1.c_str()!=s2)
  cout << "Strings sind ungleich!" << endl;
```

Listing 41: Problembehebung durch Umwandlung in C-String

Diese Vorgehensweise ist bei jeder Kombination der beiden String-Klassen zu empfehlen.

18 Wie kann in einem String ein Muster erkannt werden (Pattern Matching)?

Dieses Problem ist nicht ganz so leicht zu lösen. Zunächst sollten wir klären, was unter Mustererkennung (Pattern Matching) zu verstehen ist.

Im alltäglichen Umgang mit dem Computer stößt man häufiger auf Situationen, in denen Mustererkennung zum Einsatz kommt. Ein typisches Beispiel ist das Suchen nach Dateien. Möchte man beispielsweise auf einem Windows-System alle *jpg*-Dateien finden, dann gibt man als Maske `*.jpg` an. Der Stern ist in diesem Fall ein Platzhalter für eine beliebige Menge von beliebigen Zeichen. Sollen auch Dateien mit der Endung *jpeg* gefunden werden, könnte man `*.jp*g` schreiben. Allerdings würde eine Datei mit dem Namen *blob.jparg* ebenfalls gefunden.

Die Frage ist nun, wie eine solche Mustererkennung auf Strings anwendbar ist. Dazu muss die gewünschte Mächtigkeit der Mustererkennung zuerst definiert werden. In unserem Fall sollen folgende Funktionen verfügbar sein.

- Ein Joker ? darf verwendet werden. Er steht für genau ein beliebiges Zeichen. Das Muster »O?a« würde sowohl »Oma« als auch »Opa« erkennen, aber auch »Oha«, nicht aber »Oa« oder »Olala«.
- Es dürfen Hüllen gebildet werden. Unter Hüllen versteht man das beliebige Wiederholen eines Ausdrucks. Mit dem Zeichen * wollen wir die so genannte *Kleene'sche Hülle* oder einfach Hülle definieren. Allerdings wird der Hüllenoperator nicht wie bei der Dateisuche eingesetzt, sondern er ist an einen Ausdruck gebunden. Das Muster »ab*c« findet beliebig viele b, die zwischen a und c stehen, also beispielsweise »ac« oder »abbbc«.

- Als Ergänzung zur Kleene'schen Hülle soll auch die positive Hülle implementiert werden. Hier muss der Ausdruck mindestens einmal vorhanden sein. Der Ausdruck »ab+c« findet »abbbbbc«, nicht aber »ac«. Eine positive Hülle könnte auch mit dem normalen Hüllen-Operator definiert werden: »ab*bc« ist identisch mit »ab+c«.
- Das Klammern von Ausdrücken soll möglich sein. Der Ausdruck »Bla(bla)*h« findet Muster wie »Blah« oder »Blablablah«.
- Um Alternativen formulieren zu können, soll der Oder-Operator | zur Verfügung stehen. Der Ausdruck »O(m|p)a« findet nur »Oma« und »Opa«.
- Sollen für ein Zeichen mehrere Möglichkeiten in Betracht kommen, benutzen wir geschweifte Klammern. Der Ausdruck »{MDS}ein Kind« findet die Muster »Mein Kind«, »Dein Kind« und »Sein Kind«. Die geschweiften Klammern sind demnach eine abgekürzte Schreibweise von »(M|D|S)«. Darüber hinaus erlauben sie uns bei der technischen Umsetzung eine effizientere Realisierung.
- Zu guter Letzt soll mit den eckigen Klammern ein optionaler Ausdruck gekennzeichnet werden. Dieser Ausdruck darf einmal oder keinmal vorkommen. Der Ausdruck »[k]ein Auto« findet die Muster »ein Auto« und »kein Auto«.

Kontextfreie Grammatiken

Nachdem nun der Umfang unserer Mustererkennung geklärt ist, müssen wir die Regeln zum Aufbau eines gültigen Musters präziser formulieren. Weil es sich bei unserer Mustererkennung um eine kontextfreie Grammatik handelt, bietet sich die *Backus-Naur-Form*, kurz BNF, zur Darstellung an.

Der |-Operator soll in unserer Grammatik die geringste Bindungsstärke besitzen. Wir setzen ihn daher an oberster Stelle. Den Gesamtausdruck des definierten Musters nennen wir `expression`. In der BNF werden nicht terminale Symbole in spitze Klammern gesetzt, also `<expression>`.

Ein Ausdruck kann aus mehreren mit | verknüpften Termen oder einem einzelnen Term bestehen:

```
<expression> ::= <term> | <term>|<expression>
```

Eine solche Regel wird als *Produktion* bezeichnet. Die Formulierung von Alternativen wird in der BNF durch | zum Ausdruck gebracht. Um die Verwechslungsgefahr mit dem |-Operator. der Mustererkennung zu umgehen, stellen wir terminale Symbole, also Symbole, die genau wie in der BNF auch im Muster angegeben werden müssen, unterstrichen dar.

Diese Produktion erlaubt uns die Ableitungen <term>, <term>|<term>, <term>|<term>|<term> usw.

In Anlehnung an die üblichen Rechenoperatoren kann ein Term seinerseits aus einem oder mehreren Faktoren bestehen:

```
<term> ::= <factor> | <factor><term>
```

Diese Produktion erlaubt uns die Ableitungen <factor>, <factor><factor> usw. Zwischen Faktoren wird kein Operator definiert, denn es wäre umständlich, wenn wir bei der Suche nach »for« die einzelnen Buchstaben mit einem Operator wie * verbinden müssten (»f*o*r«).

Die beiden Produktionen zusammen erlauben uns bereits Ausdrücke wie <factor>|<factor><factor>.

Ein Faktor wiederum kann ein einzelnes Zeichen, ein geklammerter Ausdruck, eine Hülle, ein optionaler Ausdruck oder eine mit {} zusammengesetzte Auswahl sein:

```
<factor> ::= <character> | (<expression>) | <hull> | <poshull> |
             <set> | <option>
```

Beginnen wir mit den Zeichen. Als Zeichen gelten alle Zeichen, die nicht als Operatoren unserer Grammatik Verwendung finden. Diese Zeichen nennen wir <valid>. Darüber hinaus gibt es den Joker ? als beliebiges Zeichen. Um aber auch nach den Zeichen unserer Grammatik (nennen wir sie <special>) suchen zu können, benutzen wir den Operator ~. Der Ausdruck ~* beispielsweise steht für das Zeichen *.

```
<character> ::= <valid> | ? | ~<special>
```

Der Vollständigkeit halber müsste genau aufgeführt werden, welche Zeichen zu <valid> und <special> gehören. Wir wollen es aber bei der Definition belassen, dass alle Zeichen der Mustererkennung zu <special> zählen und sich alle anderen in <valid> wieder finden.

Die beiden Hüllen sind wie folgt definiert:

```
<hull>    ::= (<expression>)* | <character>*
<poshull> ::= (<expression>)+ | <character>+
```

Dabei entspricht <hull> der Kleene'schen Hülle und <poshull> der positiven Hülle. Die Hüllen selbst können entweder aus einem Zeichen oder einem Ausdruck bestehen.

Das explizite Erwähnen des Zeichens ist eigentlich überflüssig, weil <expression> auch zu einem Zeichen abgeleitet werden kann (<expression> kann ein <term> sein, der wiederum ein <factor> sein kann. Und <factor> schließlich kann ein <character> sein.)

Die zusätzliche Angabe von <character> bei den Hüllenproduktionen lässt jedoch auf eine später effektivere Implementierung schließen.

Der optionale Ausdruck kann ebenfall entweder ein einzelnes Zeichen oder ein kompletter Ausdruck sein.

```
<option> ::= [<expression>] | [<character>]
```

Eine zusammengesetzte Auswahl kann aus einem oder mehreren <valid> und <special> Zeichen bestehen, wobei die <special>-Zeichen mit dem Operator ~ angegeben werden müssen.

```
<set>      ::= {<setchars>}
<setchars> ::= ( <valid> | ~<special> ) [ <setchars> ]
```

Sehen wir uns die Grammatik noch einmal im Überblick an:

```
<expression> ::= <term>|<term>|<expression>
<term>       ::= <factor> | <factor><term>
<factor>     ::= <character> | (<expression>) | <hull> | <poshull> |
                 <set> | <option>
<character>  ::= <valid> | ? | ~<special>
<hull>       ::= (<expression>)* | <character>*
<poshull>    ::= (<expression>)+ | <character>+
<option>     ::= [<expression>] | [<character>]
<set>        ::= {<setchars>}
<setchars>   ::= ( <valid> | ~<special> ) [ <setchars> ]
```

Die Mustererkennung ist spezifiziert. Es muss nun eine Methode gefunden werden, ein beliebiges Muster in ein lauffähiges Programm umzuwandeln.

Nichtdeterministische endliche Automaten

Wir bedienen uns dazu der Automatentheorie. Jede Funktion unserer Mustererkennung wird durch einen nichtdeterministischen endlichen Automaten (kurz: NEA) dargestellt. Diese einzelnen Automaten können dann zu einem komplexeren Automaten zusammengesetzt werden, der das gewünschte Muster erkennt. Schauen wir uns als Erstes einen Automaten an, der das Muster `x` erkennt:

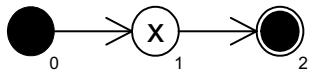


Abbildung 8: Ein Automat zur Erkennung des Musters »x«

Vom Startzustand 0 springt der Automat sofort in den Zustand 1. Diesen kann er nur verlassen, wenn das aktuelle Zeichen der Zeichenkette (zu Beginn das erste) ein »x« ist. Sollte es sich beim aktuellen Zeichen um ein »x« handeln, so springt der Automat in den Folgezustand und betrachtet das Folgezeichen. Der Folgezustand ist jedoch der Endzustand.

Erreicht der Automat den Endzustand und hat zu diesem Zeitpunkt alle Zeichen der Zeichenkette durchlaufen, dann wurde das Muster erfolgreich erkannt.

In den folgenden drei Fällen wird das Muster nicht erkannt:

1. Der Automat erreicht den Endzustand, hat aber noch nicht alle Zeichen der Zeichenkette durchlaufen.
2. Die Zeichenkette wurde bereits komplett durchlaufen, aber der Automat hat den Endzustand noch nicht erreicht.
3. Der Automat kann den aktuellen Zustand nicht verlassen, weil das mit dem aktuellen Zustand assoziierte Zeichen nicht mit dem aktuellen Zeichen der Zeichenkette übereinstimmt.

Der Automat in Abbildung 9 erkennt das Muster `for`.

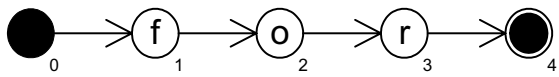


Abbildung 9: Ein Automat zur Erkennung des Musters »for«

Sollte der Automat die Zeichenkette »formel« überprüfen, dann würde er seinen Endzustand erreichen, ohne dass die Zeichenkette vollständig bearbeitet wäre.

Bei »fo« wäre die Zeichenkette durchlaufen, bevor der Automat den Endzustand erreicht hätte.

Und bei »fly« bliebe der Automat in Zustand 2 stecken, weil das »o« des Zustands nicht mit dem aktuellen Zeichen »l« der Zeichenkette übereinstimmt.

Im Folgenden werden die einzelnen Funktionen der Mustererkennung als Automaten beschrieben. Als Erstes werfen wir einen Blick auf einen Automaten, der das Muster `a(n|m)` erkennt. Er ist in Abbildung 10 dargestellt.

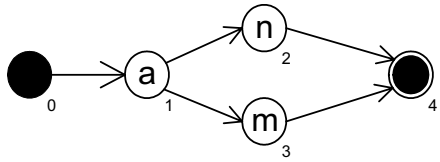


Abbildung 10: Ein Automat zur Erkennung des Musters »a(n|m) «

Alternative Möglichkeiten sind im Automaten daran zu erkennen, dass ein Zustand mehrere Folgezustände hat.

Die Kleene'sche Hülle ist in Abbildung 11 dargestellt.

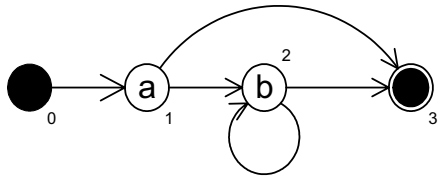


Abbildung 11: Ein Automat zur Erkennung von »ab«*

Das Besondere an der Kleene'schen Hülle im Vergleich zur positiven Hülle ist die Tatsache, dass die Kleene'sche Hülle aus keinem Element bestehen kann. Im Fall des Musters ab* wird daher auch die Zeichenkette »a« erkannt. Am Automaten erkennt man dies an dem Pfeil, der von Zustand 1 direkt zum Endzustand führt.

Bei der positiven Hülle fehlt dieser Pfeil, wie Abbildung 12 zeigt.

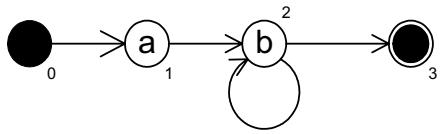


Abbildung 12: Ein Automat zur Erkennung von »ab+«

Erzeugung des Automaten

Beginnen wir nun mit der Implementierung. Zunächst erstellen wir eine Klasse `CState`, die einen Zustand des Automaten repräsentiert.

```
class CState {
public:
  enum TYPES {NOOP=0, JOKER, START, END, CHAR};
  TYPES type;
  vector<char> characters;
  vector<int> nextstates;
  CState(TYPES t) : type(t) {}
};
```

Listing 42: Die Klasse CState

Ein Zustand kann verschiedene Typen haben.

- CHAR – der Zustand, der eine Auswahl von Zeichen repräsentiert
- JOKER – der Zustand, der ein beliebiges Zeichen repräsentiert
- NOOP – als Abkürzung für No Operation der Zustand, der keinen direkten Beitrag zur Mustererkennung liefert, sondern nur aus technischen Gründen eingesetzt wird
- END – der Endzustand

Der Typ des Zustands wird im Attribut `type` abgelegt. Der Vektor `characters` enthält alle Zeichen, die den Automaten vom aktuellen Zustand in einen Folgezustand wechseln lassen. Und `nextstates` beinhaltet die Indizes aller Folgezustände des aktuellen Zustandes.

Der Automat selbst wird als Klasse `CAutomat` implementiert. Im Folgenden ist ein Überblick über den Klasseninhalt aufgeführt.

```
class CAutomat {
private:
  class CState { /* … */ };
  vector<CState> states;

  void build(const string &pattern);
```

Listing 43: Die Klasse CAutomat im Überblick

```
  int expression(string::const_iterator &i);
  int term(string::const_iterator &i);
  int factor(string::const_iterator &i);
  int option(string::const_iterator &i);
  int hull(int state) ;
  int poshull(int state);
  int character(string::const_iterator &i);
  int set(string::const_iterator &i);
  bool r_match(int state,
               string::const_iterator iter,
               string::const_iterator &end);

public:
  class EParseError {};
  CAutomat(const string &pattern);
  void showAutomat();
  bool match(const string &s,
             size_t spos=0,
             size_t len=string::npos);
};
```

Listing 43: Die Klasse CAutomat im Überblick (Forts.)

Die Klasse `CState` wurde als lokale Klasse von `CAutomat` definiert. Zusätzlich existiert noch die leere Klasse `EParseError`, die als Ausnahme geworfen wird, wenn die Umwandlung des Musters in einen Automaten fehlschlägt.

Die Klasse `CAutomat` besitzt nur ein Attribut, den Vektor `states`, der die Zustände des Automaten aufnimmt. Es stehen die nachfolgenden öffentlichen Methoden zur Verfügung.

- `showAutomat` gibt den Automat mit allen Zuständen aus.
- `match` prüft, ob eine Zeichenkette als Muster erkannt wird.

Dem Konstruktor wird eine das umzusetzende Muster enthaltende Zeichenkette übergeben.

Die privaten Methoden besitzen folgende Bedeutung:

- `build` startet die Umwandlung des als Zeichenkette vorliegenden Musters, indem sie den Startzustand aufruft und `expression` aufruft.
- `expression` setzt die Produktion von <`expression`> um. Es wird für jeden Term die Methode `term` aufgerufen.

- `term` ruft die Methode `factor` auf, wenn nicht gerade ein Ausdruck beendet wurde.
- `factor` stellt fest, ob es sich um einen geklammerten Ausdruck, einen optionalen Ausdruck, eine Auswahl von Zeichen oder ein einzelnes Zeichen handelt, und ruft entsprechend `expression`, `set`, `option` oder `character` auf. Anschließend wird auf eine eventuelle Hüllenbildung hin geprüft und unter Umständen `hull` oder `poshull` aufgerufen.
- `option` bildet einen optionalen Ausdruck.
- `hull` erzeugt eine Kleene'sche Hülle.
- `poshull` erzeugt eine positive Hülle.
- `character` erzeugt einen Zustand für ein einzelnes Zeichen.
- `set` erzeugt einen Zustand mit einer Auswahl von Zeichen.
- `r_match` ist eine rekursive Funktion, die den Automaten aufgrund der in `states` abgelegten Zustände simuliert.

expression

Werfen wir nun einen genaueren Blick auf `expression`. Aufgrund der Tatsache, dass die erzeugten Zustände hintereinander in einem Vektor abgelegt werden, gilt grundsätzlich die Regel, dass jeder Zustand zunächst einmal seinen direkten Nachfolger im Vektor als Folgezustand zugewiesen bekommt. In manchen Situationen muss dies jedoch geändert werden.

Wird ein neuer Teil des Automaten in den Vektor `states` geschrieben, dann ist der letzte Zustand des neu hinzugefügten Teils bekannt, denn er ist der letzte Zustand im Vektor. Lediglich der Anfangszustand des neuen Teils ist nicht ohne weiteres auszumachen. Deswegen liefert jede Methode, die einen neuen Teil des Automaten erzeugt, den Index des ersten Zustands des neuen Teils zurück.

Die Methode `expression` wird von `build` zum ersten Mal aufgerufen, nachdem der Anfangszustand erzeugt wurde. `expression` hat die Aufgabe, für jeden Term die Methode `term` aufzurufen und sicherzustellen, dass die von `term` erzeugten Automatenteile alle Folgezustände eines Zustands sind. Dazu legt `expression` zuerst einen technischen `NOOP`-Knoten an, der als Folgezustände alle von `term` erzeugten Automaten zugewiesen bekommt. Abbildung 13 zeigt den bisher erzeugten Automaten.

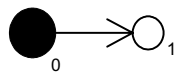


Abbildung 13: Die Situation nach Erzeugung des NOOP-Knotens

Betrachten wir nun die Umsetzung des Musters `ab|uv|yz`. Dazu werden für jeden Term `ab`, `uv` und `yz` die Methode `term` aufgerufen und die Anfangszustände der erzeugten Teile als Folgezustände des von `expression` angelegten NOOP-Knotens. Weil aber jeder Zustand für sich den Zustand mit dem nächsthöheren Index als Folgezustand besitzt, stellt sich die Situation wie in Abbildung 14 dar.

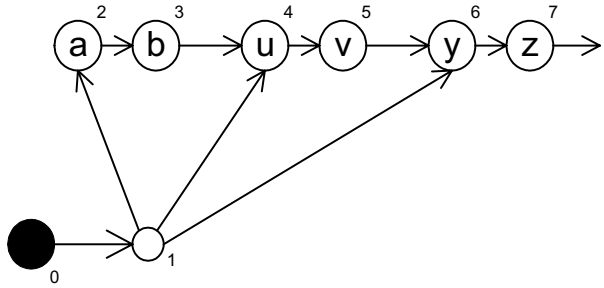


Abbildung 14: Die Situation nach dem Aufrufen von term für jeden Teilausdruck

Um diesen Automaten nun in eine vernünftige Form zu bringen, legt `expression` einen weiteren `NOOP`-Knoten an, der als Folgeknoten der einzelnen Automatenteile dienen soll. Dazu hatte `expression` nach jedem Aufruf von `term` den Endzustand des jeweiligen Automatenteils in einem Vektor gespeichert. Das Ergebnis von expression ist ein vollständiger Automat, der das Muster `ab|uv|yz` korrekt umsetzt. Abbildung 15 zeigt ihn.

Lediglich der Endzustand fehlt noch. Dies erledigt `build`, nachdem `expression` beendet wurde.

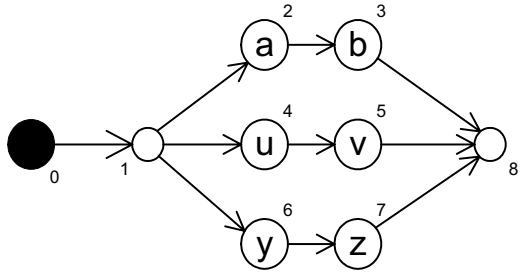


Abbildung 15: Die Situation nach Beendigung von expression

Im Folgenden ist die Funktion `expression` aufgeführt.

```
  int expression(string::const_iterator &i) {

/*
** Anlegen des Knotens, der die einzelnen Terme
** als Folgeknoten besitzen wird
*/
    int bstate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));

/*
** Der Vektor für die Aufnahme der Endzustände der
** Teilzustände
*/
    vector<int> estates;

/*
** Ersten Term umwandeln und als Folgezustand eintragen
*/
    states[bstate].nextstates.push_back(bstate+1);
    term(i);
    estates.push_back(static_cast<int>(states.size())-1);

/*
** Eventuell weitere Terme umwandeln und als Folgezustände
** eintragen
*/
    while(*i=='|') {
      states[bstate].nextstates.push_back(term(++i));
      estates.push_back(static_cast<int>(states.size())-1);
    }

/*
** Endzustand für expression anlegen
*/
    int estate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));
    states.back().nextstates.push_back(estate+1);

/*
** Den Endzustand von expression als Folgezustand der einzelnen
** Teilautomaten eintragen
```

Listing 44: Die Methode expression

```
*/
    for(vector<int>::iterator iter=estates.begin();
        iter!=estates.end();
        ++iter) {
      states[*iter].nextstates.clear();
      states[*iter].nextstates.push_back(estate);
    }
    return(bstate);
  }
```

Listing 44: Die Methode expression (Forts.)

build

Die Aufgabe der Funktion `build` wurde bereits weiter oben beschrieben. Der Vollständigkeit halber wird nun noch der dokumentierte Quellcode aufgeführt.

```
  void build(const string &pattern) {
    states.clear();

/*
** Startzustand des Automaten anlegen
*/
    states.push_back(CState(CState::NOOP));

/*
** Iterator auf erstes Zeichen der das Muster
** enthaltenden Zeichenkette anlegen
*/
    string::const_iterator i=pattern.begin();

/*
** Startzustand von expression als Folgezustand
** des Automaten-Startzustandes eintragen
*/
    states[0].nextstates.push_back(expression(i));

/*
** Endzustand anlegen
*/
```

Listing 45: Die Methode build

```
    states.push_back(CState(CState::END));
  }
```

Listing 45: Die Methode build (Forts.)

term

Die Methode `term` macht nun nichts anderes, als so lange `factor` aufzurufen, bis ein komplexer Ausdruck abgeschlossen wurde.

```
  int term(string::const_iterator &i) {

/*
** factor aufrufen für ersten Factor
*/
    int bstate=factor(i);

/*
** So lange factor aufrufen, wie kein komplexer Ausdruck
** beendet wird
*/
    while((*i!=0)&&(*i!='|')&&(*i!=')')&&(*i!=']')) factor(i);
    return(bstate);
  }
```

Listing 46: Die Methode term

factor

An der Produktion für `<factor>` in der BNF lässt sich schon erkennen, dass der Faktor auf gewisse Weise der Dreh- und Angelpunkt der Automatenumsetzung ist. Deswegen ist die Methode `factor` auch ein Herzstück der Implementierung.

Laut der Produktion kann ein Faktor geradezu alles sein: ein kompletter Ausdruck, ein optionaler Ausdruck, eine Auswahl von Zeichen oder nur ein einzelnes Zeichen. Die Methode `factor` überprüft durch Lesen eines Zeichens (look ahead), um welchen Typ es sich bei dem aktuellen Faktor handelt, und ruft die dazugehörige Methode auf.

```
  int factor(string::const_iterator &i) {
    int bstate;

/*
** Für geklammerten Ausdruck expression aufrufen
*/
    if(*i=='(') {
      bstate=expression(++i);
      if((*(i++))!=')')
        throw(EParseError());
    }

/*
** Für Auswahl von Zeichen set aufrufen
*/
    else if(*i=='{') {
      bstate=set(++i);
      if((*(i++))!='}')
        throw(EParseError());
    }

/*
** Für optionalen Ausdruck option aufrufen
*/
    else if(*i=='[') {
      bstate=option(++i);
      if((*(i++))!=']')
        throw(EParseError());
    }

/*
** Ansonsten über character einzelnes Zeichen in Zustand umwandeln
*/
    else
      bstate=character(i);

/*
** Ist der soeben bearbeitete Faktor mit einem Hüllen-Operator
** versehen, dann hull oder poshull aufrufen.
*/
    if(*i=='*') {
```

Listing 47: Die Methode factor

```
      i++;
      bstate=hull(bstate);
    }
    else if(*i=='+') {
      i++;
      bstate=poshull(bstate);
    }
    return(bstate);
  }
```

Listing 47: Die Methode factor (Forts.)

hull

Die Funktionsweise von `hull` ist wieder etwas komplizierter. Wie wir der Implementierung von `factor` entnehmen können, ist ein Ausdruck, auf den ein Hüllenoperator angewendet wird, schon komplett in Automatenzustände umgewandelt, bis `factor` feststellt, dass eine Hülle gebildet werden muss.

Nehmen wir als Beispiel den Ausdruck `x(abc)*z`. Zum Zeitpunkt, wo in `factor` der Hüllenoperator `*` entdeckt wird, ist der Automat bereits wie in Abbildung 16 aufgebaut.

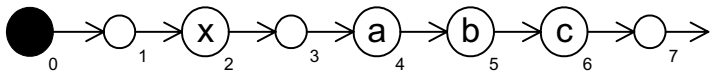


*Abbildung 16: Der Automat für x(abc)*z vor dem Hüllenoperator*

Der Zustand eins ist der technisch bedingte `NOOP`-Anfangszustand des ersten `expression`-Aufrufs. Da wegen der Klammern für `abc` die Methode `expression` erneut aufgerufen wurde, entstanden dadurch die technisch bedingten `NOOP`-Anfangs- und Endzustände 3 und 7.

Um nun eine Kleene'sche Hülle zu bilden, muss gewährleistet sein, dass der die Hülle bildende Ausdruck auch keinmal auftreten kann. Deswegen darf Zustand 2 den Zustand 3 nicht weiter als Folgezustand besitzen. Es wird dazu ein neuer Zustand 8 erzeugt, der als Folgezustand von Zustand 2 eingesetzt wird. Von Zustand 8 aus geht es entweder direkt weiter oder zu Zustand 3, um den Ausdruck der Hülle abzuarbeiten. Abbildung 17 zeigt den aktuellen Automaten.

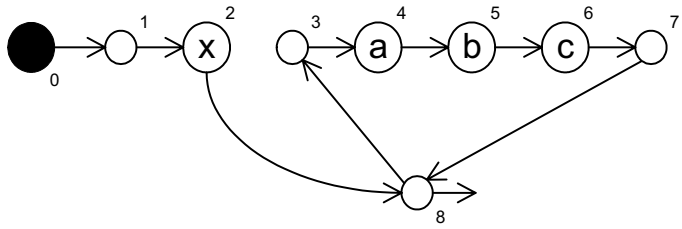


Abbildung 17: Der erste Schritt der Hüllenbildung

Nun fehlt nur noch ein technisches Detail. Wie wir bei `expression` gesehen haben, bekommt der Endzustand jedes Teilautomaten den technisch bedingten `NOOP`-Endzustand von `expression` als Folgezustand. Verweise auf vorherige Folgezustände werden gelöscht.

Im Augenblick ist bei der Hüllenbildung Zustand 8 der Endzustand des Teilautomaten. Ein Löschen der Verweise auf die aktuellen Folgezustände würde die Hüllenbildung zerstören. Es muss daher ein weiterer `NOOP`-Zustand eingefügt werden. Abbildung 18 zeigt die vollständige Hüllenbildung.

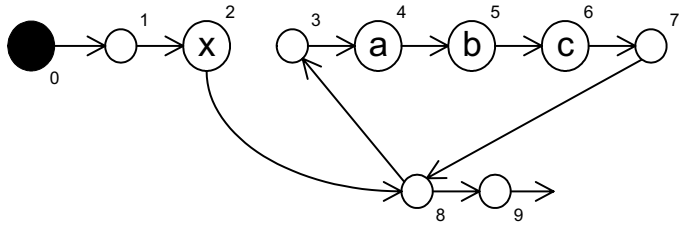


Abbildung 18: Die komplette Kleensche'sche Hülle ist gebildet

Der Vollständigkeit halber sehen Sie in Abbildung 19 den kompletten Automaten für das Muster `x(abc)*z`. Zustand 11 ist der technisch bedingte `NOOP`-Endzustand des ersten `expression`-Aufrufs.

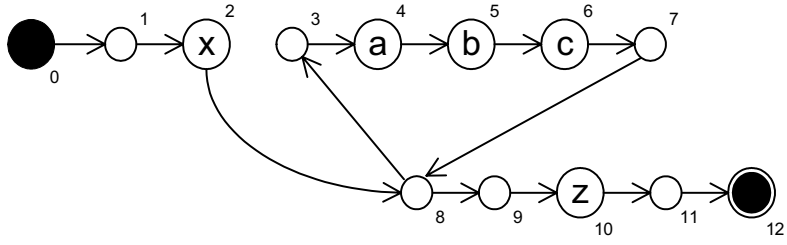


*Abbildung 19: Der komplette Automat für das Muster »x(abc)*z«*

Es scheint so, dass viele Zustände des Automaten überflüssig sind. Wir werden uns dieser Problematik zu einem späteren Zeitpunkt widmen.

Nun folgt der Quellcode der `hull`-Methode.

```
  int hull(int state) {

/*
** Ersten Zustand der Hülle sowie technisch
** bedingten Zustand erzeugen
*/
    int bstate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));
    int estate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));

/*
** Der erste Zustand der Hülle bekommt als Folgezustände
** den ersten Zustand des zur Hülle gehörenden
** Ausdrucks sowie den technisch bedingten Zustand.
*/
    states[bstate].nextstates.push_back(estate);
    states[bstate].nextstates.push_back(state);

/*
** Wie üblich bekommt der letzte Zustand den
** Zustand mit nächsthöherem Index als Folgezustand.
*/
    states[estate].nextstates.push_back(estate+1);

/*
** Um eine leere Hülle zu ermöglichen, muss der
** Vorgängerzustand des ersten Zustands des Ausdrucks
** als Folgezustand den ersten Zustand der
** Hülle bekommen.
*/
    states[state-1].nextstates.clear();
    states[state-1].nextstates.push_back(bstate);
    return(bstate);
  }
```

Listing 48: Die Methode hull

poshull

Wir wandeln das Beispiel aus dem letzten Rezept derart ab, dass die Kleene'sche Hülle durch eine positive Hülle ausgetauscht wird: `x(abc)+z`.

Die positive Hülle unterscheidet sich von der Kleene'schen Hülle nur dadurch, dass der Ausdruck der Hülle mindestens einmal auftreten muss. Für den Automaten aus Abbildung 19 hat dies zur Folge, dass die Verbindung zwischen Zustand 2 und 3 nicht gekappt zu werden braucht. Abbildung 20 zeigt den aktualisierten Automaten.

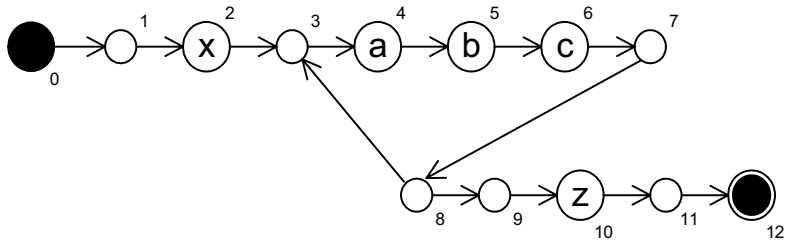


Abbildung 20: Der Automat für das Muster »x(abc)+z«

Der Quellcode von `poshull` sieht wie folgt aus:

```
  int poshull(int state) {

/*
** Ersten Zustand der Hülle sowie technisch
** bedingten Zustand erzeugen
*/
    int bstate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));
    int estate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));

/*
** Der erste Zustand der Hülle bekommt als Folgezustände
** den ersten Zustand des zur Hülle gehörenden
** Ausdrucks sowie den technisch bedingten Zustand.
*/
    states[bstate].nextstates.push_back(estate);
    states[bstate].nextstates.push_back(state);
```

Listing 49: Die Methode poshull

```
/*
** Wie üblich bekommt der letzte Zustand den
** Zustand mit nächsthöherem Index als Folgezustand.
*/
    states[estate].nextstates.push_back(estate+1);

/*
** Der Folgezustand des Vorgängerzustands des ersten
** Zustands des Hüllenausdrucks bekommt explizit den
** ersten Zustand des Hüllenausdrucks als Folgezustand.
*/
    states[state-1].nextstates.clear();
    states[state-1].nextstates.push_back(state);
    return(bstate);
  }
```

Listing 49: Die Methode poshull (Forts.)

option

Die Methode `option` implementiert den in eckigen Klammern geschriebenen optionalen Ausdruck.

Zunächst wird wegen der einfacheren Verwaltung ein `NOOP`-Zustand als Startzustand des zu bildenden Teilautomaten angelegt. Dann wird über den Aufruf von `expression` der Ausdruck innerhalb der eckigen Klammern in einen Automaten umgewandelt.

Daran wird ein `NOOP`-Zustand als Endzustand des Teilautomaten angehängt.

Zum Schluss bekommt der Startzustand des Teilautomaten den ersten Zustand des Ausdrucks und den Endzustand als Folgezustand.

Der Quellcode von `option` stellt sich folgendermaßen dar:

```
  int option(string::const_iterator &i) {

/*
** Technisch bedingten NOOP-Zustand als
** Startzustand des Teilautomaten anlegen
*/
    int bstate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));
```

Listing 50: Die Methode option

```
/*
** Optionalen Ausdruck in Automaten umwandeln
*/
    expression(i);

/*
** Technisch bedingten NOOP-Zustand als
** Startzustand des Teilautomaten anlegen
*/
    int estate=static_cast<int>(states.size());
    states.push_back(CState(CState::NOOP));

/*
** Verbindungen zwischen Start- und Endzustand
** des Teilautomaten und dem optionalen Ausdruck
** herstellen
*/
    states[bstate].nextstates.push_back(estate);
    states[bstate].nextstates.push_back(bstate+1);
    states[estate].nextstates.push_back(estate+1);
    return(bstate);
  }
```

Listing 50: Die Methode option (Forts.)

character

Die Methode `character` ist dafür verantwortlich, ein einzelnes Zeichen in einen Zustand des Automaten umzuwandeln.

Als einzelnes Zeichen gelten die durch `<valid>` definierten Zeichen, der Joker und die `<special>`-Zeichen, die mit dem Operator ~ eingeleitet werden.

```
  int character(string::const_iterator &i) {
    int bstate=static_cast<int>(states.size());

/*
** Wenn es sich bei Zeichen um Joker handelt, dann
** speziellen JOKER-Zustand erzeugen.
*/
```

Listing 51: Die Methode character

```
    if(*i=='?') {
      states.push_back(CState(CState::JOKER));
    } else {

/*
** Handelt es sich um das Zeichen ~ (Beginn eines
** <special>-Zeichens), dann überspringen
*/
      if(*i=='~') ++i;

/*
** CHAR-Zustand für Zeichen erzeugen
*/
      states.push_back(CState(CState::CHAR));
      states.back().characters.push_back(*i);
    }

/*
** Wie üblich bekommt der letzte Zustand den
** Zustand mit nächsthöherem Index als Folgezustand.
*/
    states.back().nextstates.push_back(bstate+1);
    ++i;
    return(bstate);
  }
```

Listing 51: Die Methode character (Forts.)

set

Die Methode `set` bringt mehrere Zeichen in einem Zustand unter. Hätten wir die Schreibweise mit den geschweiften Klammern (z.B.`{asdf}`) nicht in unsere Mustererkennung eingebaut, müsste der Ausdruck über `|` formuliert werden: `(a|s|d|f)`. Letztere Variante benötigt vier Zustände und erzeugt demnach einen komplexeren Automaten als der Ansatz über {}.

Der Quellcode von `set` ist im Folgenden aufgeführt.

```
  int set(string::const_iterator &i) {

/*
** Der Zustand für die Auswahl von Zeichen
```

```
** wird erzeugt.
*/
    int bstate=static_cast<int>(states.size());
    states.push_back(CState(CState::CHAR));

/*
** Alle Zeichen innerhalb der geschweiften Klammern
** werden in den Zustand übertragen.
*/
    while(*i!='}') {
      if(*i=='~') ++i;
      states.back().characters.push_back(*i);
      i++;
    }

/*
** Wie üblich bekommt der letzte Zustand den
** Zustand mit nächsthöherem Index als Folgezustand.
*/
    states.back().nextstates.push_back(bstate+1);
    return(bstate);
  }
```

showAutomat

Die Methode `showAutomat` listet die einzelnen Zustände des Automaten mit ihren Folgezuständen auf.

```
  void showAutomat() {
    size_t maxwidth=0;
    for(vector<CState>::iterator iter=states.begin();
        iter!=states.end();
        ++iter)
      if(maxwidth<iter->characters.size())
        maxwidth=iter->characters.size();

    string empty(255,' ');
    for(size_t i=0; i<states.size(); i++) {
      cout << setw(3) << static_cast<int>(i) << " ";
      if(states[i].type==CState::NOOP)
```

Listing 52: Die Methode showAutomat

```
        cout << "NOP";
      if (states[i].type==CState::JOKER)
        cout << "JOK";
      if (states[i].type==CState::START)
        cout << "STA";
      if (states[i].type==CState::END)
        cout << "END";
      if (states[i].type==CState::CHAR) {
        cout << "CHR ";
        cout<<empty.substr(0,maxwidth-states[i].characters.size());
        for(vector<char>::iterator citer=
            states[i].characters.begin();
            citer!=states[i].characters.end();
            ++citer)
          cout << *citer;
      }
      else
        cout << empty.substr(0,maxwidth+1);

      for(vector<int>::iterator siter=states[i].nextstates.begin();
          siter!=states[i].nextstates.end();
          ++siter)
        cout << setw(3) << *siter;
      cout << endl;
    }
  }
```

Listing 52: Die Methode showAutomat (Forts.)

Konstruktor

Zum Schluss fehlt nur noch der Konstruktor. Er ist denkbar einfach aufgebaut, ruft er doch nur die Funktion `build` auf, die den Automaten erstellt.

```
  CAutomat(const string &pattern) {
    build(pattern);
  }
```

Listing 53: Der Konstruktor

Minimierung des Automaten

Betrachten wir einmal den Automaten, der aus dem Muster `ab|(w(xy)*z)|c` erzeugt wird. Die Methode `showAutomat` liefert folgende Ausgabe:

```
 0 STA    1
 1 NOP    2  4 14
 2 CHR a  3
 3 CHR b 15
 4 NOP    5
 5 CHR w 10
 6 NOP    7
 7 CHR x  8
 8 CHR y  9
 9 NOP   10
10 NOP   11  6
11 NOP   12
12 CHR z 13
13 NOP   15
14 CHR c 15
15 NOP   16
16 END
```

*Listing 54: Ausgabe von showAutomat für Muster »ab|(w(xy)*z)|c«*

Die grafische Darstellung des Automaten ist in Abbildung 21 zu sehen.

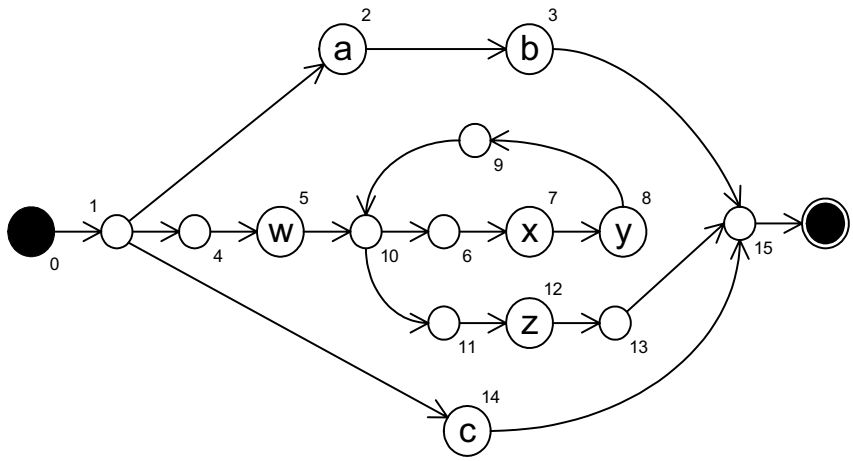


*Abbildung 21: Der Automat für »ab|(w(xy)*z)|c«*

minimalize

Es fällt auf, dass die NOOP-Zustände eigentlich unnötig sind, denn ein Zustand, der einen solchen NOOP-Zustand als Folgezustand hat, könnte gleich die Folgezustände des NOOP-Zustandes als eigene Folgezustände besitzen. Auf diese Weise lassen sich alle NOOP-Zustände aus dem Automaten entfernen.

Diesen Ansatz verfolgt die Methode minimalize. Abgesehen vom Startzustand durchläuft sie alle Zustände des Automaten und sucht NOOP-Zustände mit einem Folgezustand.

Wird ein solcher Zustand gefunden, dann wird er gelöscht und alle Verweise auf diesen Zustand werden auf den Folgezustand des gelöschten Zustands geändert.

Es ist zu beachten, dass sich durch das Löschen eines Zustands im Vektor die Indizes aller Zustände mit einem höheren Index als dem gelöschten Zustand um eins vermindern. Deswegen müssen alle Verweise auf diese Zustände entsprechend angepasst werden.

Im Folgenden ist die Methode minimalize aufgeführt.

```
  void minimalize() {

/*
** Alle Zustände des Automaten durchlaufen
*/
    int idx=0;
    vector<CState>::iterator iter=states.begin();
    while(iter!=states.end()) {

/*
** Handelt es sich um einen NOOP-Zustand?
*/
      if((iter->type==CState::NOOP)) {

/*
** Alle Knoten durchlaufen
*/
        for(vector<CState>::iterator liter=states.begin();
            liter!=states.end();
            ++liter) {
          if(iter==liter)
```

Listing 55: Die Methode minimalize

```
            continue;

/*
** Ist ein Folgezustand der zu löschende NOOP-Knoten?
*/
          vector<int>::iterator fpos=
                           std::find(liter->nextstates.begin(),
                           liter->nextstates.end(),
                           idx);
          if(fpos!=liter->nextstates.end()) {

/*
** Verweis auf zu löschenden NOOP-Knoten löschen und
** stattdessen Folgezustände des zu löschenden NOOP-
** Knotens übernehmen
*/
            fpos=liter->nextstates.erase(fpos);
            copy(iter->nextstates.begin(),
                 iter->nextstates.end(),
                 inserter(liter->nextstates,fpos));
          }

/*
** Alle Folgezustände durchlaufen
*/
          for(vector<int>::iterator siter=liter->nextstates.begin();
              siter!=liter->nextstates.end();
              ++siter)

/*
** Alle Verweise auf Zustände mit Index > idx
** auf nächstniederen Index setzen
*/
            if(*siter>idx)
              (*siter)--;

        }

/*
** NOOP-Knoten löschen
*/
```

Listing 55: Die Methode minimalize (Forts.)

```
        iter=states.erase(iter);
      }
      else {
        iter++;
        idx++;
      }
    }
  }
```

Listing 55: Die Methode minimalize (Forts.)

Wird `minimalize` auf den aus dem Muster `ab|(w(xy)*z)|c` erzeugten Automaten angesetzt, dann erhalten wir eine minimale Version, die `showAutomat` folgendermaßen ausgibt:

```
0 STA   1  3  7
1 CHR a  2
2 CHR b  8
3 CHR w  6  4
4 CHR x  5
5 CHR y  6  4
6 CHR z  8
7 CHR c  8
8 END
```

*Listing 56: Ausgabe des minimierten Automaten aus Muster »ab|(w(xy)*z)|c«*

Die grafische Darstellung des Automaten finden Sie in Abbildung 22.

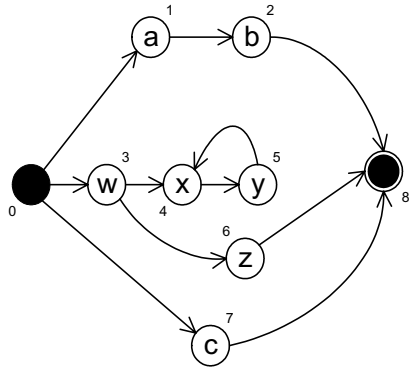


*Abbildung 22: Der minimierte Automat für »ab|(w(xy)*z)|c«*

Simulation des Automaten

Aus dem als Zeichenkette vorliegenden Muster wurde in den vorangegangenen Rezepten ein nichtdeterministischer endlicher Automat erzeugt. Nun muss dem Automaten Leben eingehaucht werden.

match

Wir implementieren eine Methode `match`, der ein String übergeben wird. Optional kann über Start und Länge ein Teilstring definiert werden.

Dann wird die rekursive Methode `r_match` aufgerufen. Sie bekommt den zu bearbeitenden Zustand (zu Beginn Startzustand 0) und als Iteratoren das aktuelle Zeichen und das Ende der Zeichenkette übergeben.

```
bool match(const string &s,
           size_t spos=0,
           size_t len=string::npos) {
  if((len!=string::npos)&&((spos+len)>s.length()))
    return(false);          // Bereich überschritten
  return(r_match(0,
                 s.begin()+spos,
                (len!=string::npos)?s.begin()+spos+len:s.end()));
}
```

Listing 57: Die Methode match

r_match

Die Methode `r_match` wertet den aktuellen Zustand des Automaten aus.

- Für einen `CHAR`-Zustand wird überprüft, ob das aktuelle Zeichen der Zeichenkette in den Zeichen des Zustandes enthalten ist. Wenn nicht, dann wurde das Muster nicht erkannt. Wenn ja, dann wird `r_match` für alle Folgezustände mit dem nächsten Zeichen der Zeichenkette aufgerufen.
- Für den `JOKER`-Zustand braucht kein Zeichen überprüft zu werden, da jedes Zeichen gültig ist. `r_match` wird für alle Folgezustände mit dem nächsten Zeichen der Zeichenkette aufgerufen.
- Für die `NOOP`-Zustände und den `START`-Zustand brauchen keine Zeichen überprüft zu werden. Es wird lediglich `r_match` für alle Folgezustände mit dem aktuellen Zeichen aufgerufen.

Sollte der Automat seinen Endzustand erreicht haben und die Zeichenkette ist komplett bearbeitet, dann wurde das Muster erkannt.

Wurde der Endzustand erreicht, ohne dass alle Zeichen der Zeichenkette vom Automaten bearbeitet wurden, dann wurde das Muster nicht erkannt.

```
  bool r_match(int state,
               string::const_iterator iter,
               string::const_iterator &end) {

/*
** Hat die Zeichenkette ihr Ende erreicht?
*/
    if(iter==end) {

/*
** Hat der Automat seinen Endzustand erreicht?
** => Muster wurde erkannt.
*/
      if(states[state].type==CState::END)
        return(true);

/*
** Muss der Automat noch Zeichen auswerten?
** => Muster wurde  nicht erkannt.
*/
      if(states[state].type!=CState::NOOP)
        return(false);
    }

/*
** Status auswerten
*/
    switch(states[state].type) {

/*
** CHAR-Zustand
*/
      case CState::CHAR:

/*
** Das aktuelle Zeichen der Zeichenkette wird mit
** den Zeichen des Zustands verglichen.
** Stimmt es nicht überein, dann wurde das Muster
```

Listing 58: Die Methode r_match

```
** nicht erkannt.
*/
        if(std::find(states[state].characters.begin(),
                     states[state].characters.end(),
                    *iter)==states[state].characters.end()) {
          return(false);
        }
        else {

/*
** Bei Übereinstimmung wird r_match
** rekursiv für jeden Folgezustand mit dem nächsten
** Zeichen der Zeichenkette aufgerufen.
*/
          for(vector<int>::iterator siter=
                          states[state].nextstates.begin();
              siter!=states[state].nextstates.end();
              ++siter)
            if(r_match(*siter,iter+1,end))
              return(true);
        }
          return(false);

/*
** NOOP- und START-Zustand
*/
      case CState::START:
      case CState::NOOP:

/*
** r_match wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                         states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_match(*siter,iter,end))
            return(true);
        return(false);
```

Listing 58: Die Methode r_match (Forts.)

```
/*
** JOKER-Zustand (Beliebiges Zeichen)
*/
      case CState::JOKER:

/*
** r_match wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                        states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_match(*siter,iter+1, end))
            return(true);

        return(false);
    }
    return(false);
  }
```

Listing 58: Die Methode r_match (Forts.)

Um die Einsatzmöglichkeiten zu erweitern, liegen die Methoden `match` und `r_match` in der endgültigen Fassung als Templates vor. Die Signaturen sehen dann wie folgt aus:

```
template<typename Type>
bool match(Type &s, size_t spos=0, size_t len=Type::npos) const;

template<typename Type>
bool r_match(int state, Type iter, Type &end) const;
```

Listing 59: match und r_match als Templates

Die Klasse `CAutomat` finden Sie auf der CD in den `Cautomat`-Dateien.

Zum Schluss noch eine `main`-Funktion, die den Automaten aus einem Muster erzeugt, ihn nach der Erstellung in seiner Urform und in seiner minimierten Form ausgibt und das Muster dann auf eine Zeichenkette ansetzt.

```
#include "CAutomat.h"
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  try {
    cout << "Bitte Muster eingeben:";
    string pattern;
    getString(pattern,100);
    cout << "Bitte Teststring eingeben:";
    string s;
    getString(s,100);

    CAutomat a(pattern);
    a.showAutomat();
    cout << "-----------------" << endl;
    a.minimalize();
    a.showAutomat();
    cout << "-----------------" << endl;
    cout << "Gesucht wird in " << s << endl;

    if(a.match(s))
      cout << "Muster erkannt!" << endl;
    else
      cout << "Muster nicht erkannt!" << endl;
  }
  catch(CAutomat::EParseError) {
    cout << "Parser-Fehler!" << endl;
  }
}
```

Listing 60: Eine main-Funktion zum Testen

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0209.cpp*. Sie macht von `getString` aus Rezept 11 Gebrauch. Ein Beispiel mit dem ursprünglich erzeugten und anschließend minimierten Automaten sehen Sie in Abbildung 23.

Der Einfachheit halber wird der Aufruf der `minimalize`-Methode für zukünftige Anwendungen direkt im Konstruktor integriert.

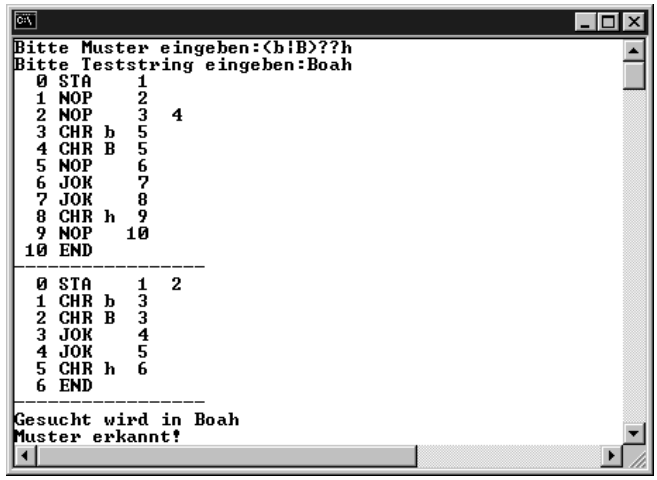


Abbildung 23: Ein Beispiel zu CAutomat

rebuild

Um ein bereits existierendes `CAutomat`-Objekt mit einem neuen Muster füttern zu können, wird zum Schluss noch die `rebuild`-Methode implementiert:

```
void rebuild(const std::string &pattern) {
  build(pattern);
  minimalize();
}
```

Listing 61: Die Methode rebuild

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`:

Die Stream-Operatoren müssen als Freunde der Klasse deklariert werden.

Statten wie zunächst die Klasse `CState` mit Operatoren aus:

```
friend CBinaryOStream &operator<<(CBinaryOStream &os,
                                  const CState &s){
  os.write(&s.m_type, sizeof(s.m_type));
```

Listing 62: Die Stream-Operatoren von CState

```
  os << s.m_characters;
  os << s.m_nextstates;
  return(os);
}
friend CBinaryIStream &operator>>(CBinaryIStream &is, CState &s){
  is.read(&s.m_type, sizeof(s.m_type));
  is >> s.m_characters;
  is >> s.m_nextstates;
  return(is);
}
```

Listing 62: Die Stream-Operatoren von CState (Forts.)

Fehlen zum Schluss noch die Operatoren für `CAutomat`:

```
CBinaryOStream &operator<<(CBinaryOStream &os, const CAutomat &a) {
  os << a.m_states;
  return(os);
}

CBinaryIStream &operator>>(CBinaryIStream &is, CAutomat &a) {
  is >> a.m_states;
  return(is);
}
```

Listing 63: Die Stream-Operatoren von CAutomat

19 Wie kann in einem String nach einem Muster gesucht werden (Pattern Matching)?

Mit dem aus Rezept 18 erzeugten Automaten zur Erkennung eines Musters lässt sich recht einfach nach einem Muster suchen.

find

Die Methode `find` wird zu `CAutomat` hinzugefügt. Ihr werden ein String und die Startposition der Suche übergeben.

Die rekursive Methode `r_find` wird mit dem Startzustand des Automaten und der Startposition der Suche als Iterator aufgerufen. Sollte die Mustererkennung erfolglos gewesen sein, dann wird die Startposition der Mustererkennung um eins verschoben und `r_find` erneut aufgerufen.

Dies geschieht so lange, bis entweder von `r_find` eine erfolgreiche Mustererkennung gemeldet wurde oder die Startposition das Ende des Strings erreicht hat.

Hier zunächst die Methode `find`:

```
size_t find(const string &s, size_t spos=0) {
  for(string::const_iterator iter=s.begin()+spos;
      iter!=s.end();
      ++iter,++spos)
    if(r_find(0, iter))
      return(spos);

  return(string::npos);
}
```

Listing 64: Die Methode find

Die Methode `r_find` ist eine Abwandlung der Methode `r_match`. `r_find` benötigt jedoch keine Endemarkierung der Zeichenkette, weil im Zweifel die gesamte Zeichenkette durchsucht wird. Auch besitzt `r_find` andere Kriterien zur Entscheidung, ob eine Mustererkennung erfolgreich war.

r_find

Im Anschluss ist die Methode `r_find` aufgeführt.

```
bool r_find(int state, string::const_iterator iter) {

/*
** Hat der Automat seinen Endzustand erreicht?
** => Muster wurde erkannt.
*/
    if(states[state].type==CState::END)
      return(true);

/*
```

Listing 65: Die Methode r_find

```
** Ist das Ende der Zeichenkette erreicht, es müssen
** aber noch CHAR-Zustände des Automaten ausgewertet
** werden => Muster wurde  nicht erkannt.
*/
    if((*iter==0)&&(states[state].type!=CState::NOOP))
      return(false);

/*
** Status auswerten
*/
    switch(states[state].type) {

/*
** CHAR-Zustand
*/
      case CState::CHAR:

/*
** Das aktuelle Zeichen der Zeichenkette wird mit
** den Zeichen des Zustands verglichen.
** Stimmt es nicht überein, dann wurde das Muster
** nicht erkannt.
*/
        if(std::find(states[state].characters.begin(),
                     states[state].characters.end(),
                     *iter)==states[state].characters.end()) {
          return(false);
        }
        else {

/*
** Bei Übereinstimmung wird r_find
** rekursiv für jeden Folgezustand mit dem nächsten
** Zeichen der Zeichenkette aufgerufen.
*/
          for(vector<int>::iterator siter=
                                  states[state].nextstates.begin()
              ;siter!=states[state].nextstates.end();
              ++siter)
            if(r_find(*siter,iter+1))
              return(true);
```

Listing 65: Die Methode r_find (Forts.)

```
        }
          return(false);

/*
** NOOP- und START-Zustand
*/
      case CState::START:
      case CState::NOOP:

/*
** r_find wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                                  states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_find(*siter,iter))
            return(true);
        return(false);

/*
** JOKER-Zustand (Beliebiges Zeichen)
*/
      case CState::JOKER:

/*
** r_find wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                                  states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_find(*siter,iter+1))
            return(true);

        return(false);
    }
    return(false);
  }
```

Listing 65: Die Methode r_find (Forts.)

rfind

Um genau wie bei den üblichen Stringsuchfunktionen auch vom Ende des Strings beginnend suchen zu können wollen wir noch eine Methode `rfind` implementieren:

```
  size_t rfind(const string &s, size_t spos=string::npos) {
    if (spos==string::npos)
      spos=s.length()-1;
    for(string::const_iterator iter=s.begin()+spos; iter!=s.end(); --iter,--
spos)
      if(r_find(0, iter))
        return(spos);

    return(string::npos);
  }
```

Listing 66: Die Methode rfind

Wie auch die ursprünglichen Methoden wollen wir `find`, `rfind` und `r_find` als Templates definieren. Hier sind die Methodenköpfe:

```
template<typename Type>
bool r_find(int state, Type iter) const{

template<typename Type>
size_t find(Type &s, size_t spos=0) const;

template<typename Type>
size_t rfind(Type &s, size_t spos=Type::npos) const;
```

Listing 67: find und r_find als Templates

Die Klasse `CAutomat` finden Sie auf der CD in den `CAutomat`-Dateien.

Die folgende `main`-Funktion eignet sich zum Testen:

```
#include "CAutomat.h"
#include "StringFunctions.h"
#include <iostream>
```

Listing 68: Eine main-Funktion zum Testen

```
using namespace std;
using namespace Codebook;

int main() {
  try {
    cout << "Bitte Muster eingeben:";
    string pattern;
    getString(pattern,100);
    cout << "Bitte Teststring eingeben:";
    string s;
    getString(s,100);

    CAutomat a(pattern);

    size_t pos=a.find(s);
    if(pos==string::npos)
      cout << "Nicht gefunden!" << endl;
    else
      cout << "Gefunden an Position " <<
              static_cast<int>(pos) << endl;
  }
  catch(CAutomat::EParseError) {
    cout << "Parser-Fehler!" << endl;
  }
}
```

Listing 68: Eine main-Funktion zum Testen (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0210.cpp*. Sie macht von `getString` aus Rezept 11 Gebrauch.

20 Wie findet man das n-te Vorkommen eines Teilstrings?

Um dieses Problem zu lösen, muss eigentlich nur die Methode `find` von `string` mehrmals aufgerufen werden, wobei die Startposition der Suche entsprechend angepasst wird.

multiple_find

Wir implementieren dazu die Funktion `multiple_find`, die fünf Parameter besitzt, wobei die letzten beiden Standard-Argumente besitzen, also optional sind.

- `s` ist der String, in dem gesucht wird.
- `sub` ist der Teilstring, der gesucht wird.
- `nth` definiert, das wievielte Vorkommen gesucht werden soll.
- `start` definiert die Position in `s`, ab der gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Bevor wir uns der genauen Bedeutung von `complete` zuwenden, werfen wir erst einen Blick auf den Quellcode:

```
size_t multiple_find(const string &s,
                     const string &sub,
                     int nth,
                     size_t start=0,
                     bool complete=true) {

/*
** Ungültige Werte?
*/
  if(nth<1||start<0)
    return(string::npos);

/*
** Erstes Vorkommen ab Position start finden
*/
  size_t fpos=s.find(sub,start);
  nth--;

/*
** Bis zum n-ten Vorkommen durchlaufen
*/
  while(nth--) {

/*
** Complete==true?
```

Listing 69: Die Funktion multiple_find

```
**  => Suchposition um eins verschieben
** complete==false?
**  => Suchposition um Länge des Teilstrings verschieben
*/
    fpos+=(complete)?sub.length():1;
    fpos=s.find(sub,fpos);

    if(fpos==string::npos)
      return(fpos);
  }
  return(fpos);
}
```

Listing 69: Die Funktion multiple_find (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

Wie bereits in der Parameterübersicht beschrieben, wird über den Parameter `complete` festgelegt, ob nur komplette Vorkommnisse von `sub` gesucht werden sollen. Betrachten wir folgenden Aufruf:

```
string s="abababababab";
string sub="abab";
size_t pos=multiple_find(s,sub,3,0,false);
```

Listing 70: Ein Aufruf von multiple_find mit complete=false

Die Variable `pos` besitzt den Wert 4, weil nicht nach kompletten Vorkommnissen von `sub` gesucht wird. Betrachten wir dazu Abbildung 24, die den Sachverhalt grafisch darstellt.

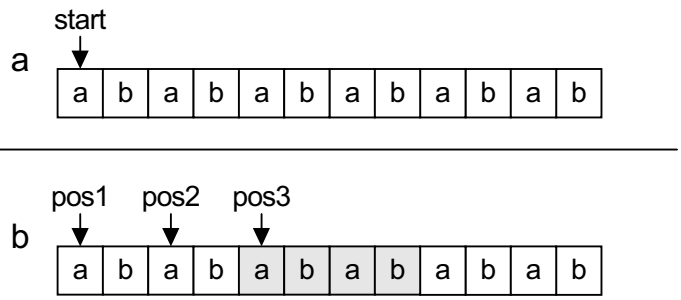


Abbildung 24: multiple_find mit complete=false

Nachdem »abab« an Zeichenposition 0 zum ersten Mal gefunden wurde, ist der zweite Fund gleich bei Position 2, obwohl die ersten beiden Zeichen des zweiten Fundes noch zum ersten Fund gehören.

Mit `complete` auf `true` gesetzt, würde der komplette erste Fund übersprungen, sodass »abab« erst an Zeichenposition 4 zum zweiten Mal und an Zeichenposition 8 zum dritten Mal gefunden wird. Abbildung 25 stellt diese Situation dar.

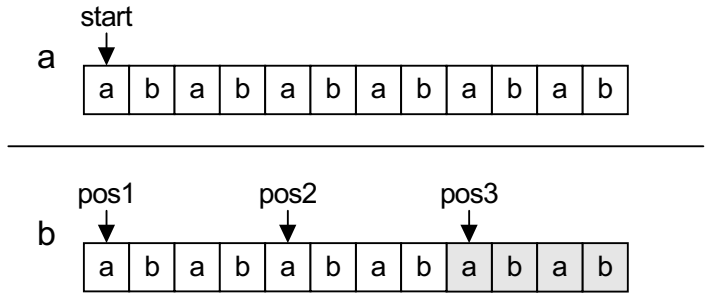


Abbildung 25: find_multiple mit complete=true

21 Wie findet man das n-te Vorkommen eines Teilstrings von hinten?

Um die gleiche Funktionaliät wie `multiple_find` aus Rezept 20, nur für eine Suche von hinten nach vorne zu erhalten, wird die Funktion `multiple_rfind` definiert.

multiple_rfind

Sie besitzt folgende Parameter:

- `s` ist der String, in dem gesucht wird.
- `sub` ist der Teilstring, der gesucht wird.
- `nth` definiert, das wievielte Vorkommen gesucht werden soll.
- `start` definiert die Position in `s`, ab der rückwärts gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Der Quellcode sieht wie folgt aus:

```
size_t multiple_rfind(const string &s,
                      const string &sub,
                      int nth,
                      size_t start=string::npos,
                      bool complete=true) {

/*
** Ungültige Werte?
*/
  if(nth<1||start<0)
    return(string::npos);

/*
** Letztes Vorkommen ab Position start finden
*/
  size_t fpos=s.rfind(sub,start);
  nth--;

/*
** Bis zum n-ten Vorkommen zurücklaufen
*/
  while(nth--) {

/*
** Neue Startposition < 0?
**  => Nicht gefunden
*/
    if((fpos<static_cast<size_t>((complete)?sub.length():1)))
      return(string::npos);

/*
** Complete==true?
**  => Suchposition um eins zurücksetzen
** complete==false?
**  => Suchposition um Länge des Teilstrings zurücksetzen
*/
    fpos-=(complete)?sub.length():1;
    fpos=s.rfind(sub,fpos);

    if(fpos==string::npos)
      return(fpos);
```

Listing 71: Die Funktion multiple_rfind

```
    }
    return(fpos);
}
```

Listing 71: Die Funktion multiple_rfind (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

22 Wie findet man das n-te Vorkommen eines Musters in einem String?

Um dieses Problem zu lösen, greifen wir auf die Klasse `CAutomat` aus Rezept 18 zurück, die ein in einer Zeichenkette vorliegendes Muster in einen nichtdeterministischen endlichen Automaten (NEA) umwandelt und diesen Automaten dann simuliert.

multiple_find

Diese Klasse erweitern wir um eine Methode `multiple_find`, die die bereits in Rezept 20, »Wie findet man das n-te Vorkommen eines Teilstrings? « gelöste Problematik an die Mustererkennung anpasst.

Die Methode `multiple_find` besitzt folgende Parameter:

- `s` ist der String, in dem gesucht wird.
- `nth` definiert, das wievielte Vorkommen gesucht werden soll.
- `start` definiert die Position in `s`, ab der gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Ein zusätzliches Problem tritt wegen des Musters auf. Während in Rezept 20 der zu suchende Teilstring immer eine konstante Länge hatte, kann das gefundene Muster in seiner Länge variabel sein. Beispielsweise könnte das Muster `a*` auf einen String »aaaaaa« angewandt von »«, über »aa« bis hin zu »aaaaaa« alle Muster finden.

Der Automat aus Rezept 19 wurde allerdings stillschweigend so entworfen, dass er bei Hüllenbildung und optionalen Ausdrücken immer das kürzeste Muster erkennt.

Schauen wir uns die Methode `multiple_find` einmal an:

```
  size_t multiple_find(const string &s,
                       int nth,
                       size_t start=0,
                       bool complete=true) {

/*
** Ungültige Werte?
*/
    if(nth<1||start<0)
      return(string::npos);

/*
** Bis zum n-ten Vorkommen durchlaufen
*/
    while(nth--) {
      string::const_iterator iter, end;

/*
** Position im String suchen, an der Muster erkannt wird
*/
      for(iter=s.begin()+start; iter!=s.end(); ++iter,++start)
        if(r_find(0, iter,end))
          break;

/*
** Wurde Muster nicht gefunden?
*/
      if(iter==s.end())
        return(string::npos);

/*
** n-te Position erreicht?
*/
      if(!nth)
        return(start);

/*
** Complete==true?
**  => Suchposition um eins verschieben
** complete==false?
**  => Suchposition um Länge des Teilstrings verschieben
```

Listing 72: Die Methode multiple_find

```
*/
      start+=(complete)?distance(iter, end):1;
    }
    return(string::npos);
  }
```

Listing 72: Die Methode multiple_find (Forts.)

Es fällt auf, dass `multiple_find` eine andere Version der Methode `r_find` benötigt als die bereits implementierte Methode `find`.

Das liegt in der Tatsache begründet, dass `r_find` zurückliefern muss, welche Zeichenlänge das vom Automaten erkannte Muster tatsächlich war. Und weil `r_find` bisher mit Iteratoren gearbeitet hat, bekommt die zweite Fassung einen dritten Parameter `end`, der bei Beendigung des Automaten die Iteratorposition auf Zeichen hinter dem erkannten Muster aufnimmt.

Mit Hilfe der Funktion `distance` aus der Standardbibliothek wird dann die Anzahl der zwischen den Iteratoren `iter` und `end` liegenden Zeichen bestimmt.

r_find

Im Folgenden ist die zusätzliche Variante von `r_find` aufgeführt:

```
  bool r_find(int state,
              string::const_iterator iter,
              string::const_iterator &end) {

/*
** Hat der Automat seinen Endzustand erreicht?
** => Muster wurde erkannt..
*/
    if(states[state].type==CState::END) {
      end=iter;
      return(true);
    }

/*
** Ist das Ende der Zeichenkette erreicht, es müssen
** aber noch CHAR-Zustände des Automaten ausgewertet
** werden => Muster wurde  nicht erkannt.
```

Listing 73: Die Methode r_find

```
*/
    if((*iter==0)&&(states[state].type!=CState::NOOP))
      return(false);

/*
** Status auswerten
*/
    switch(states[state].type) {

/*
** CHAR-Zustand
*/
      case CState::CHAR:

/*
** Das aktuelle Zeichen der Zeichenkette wird mit
** dem Zeichen des Zustands verglichen.
** Stimmt es nicht überein, dann wurde das Muster
** nicht erkannt.
*/
        if(std::find(states[state].characters.begin(),
           states[state].characters.end(),
           *iter)==states[state].characters.end()) {
          return(false);
        }
        else {

/*
** Bei Übereinstimmung wird rfind
** rekursiv für jeden Folgezustand mit dem nächsten
** Zeichen der Zeichenkette aufgerufen.
*/
          for(vector<int>::iterator siter=
                          states[state].nextstates.begin();
              siter!=states[state].nextstates.end();
              ++siter)
            if(r_find(*siter,iter+1,end))
              return(true);
        }
          return(false);
```

Listing 73: Die Methode r_find (Forts.)

```
/*
** NOOP- und START-Zustand
*/
      case CState::START:
      case CState::NOOP:

/*
** rfind wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                        states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_find(*siter,iter,end))
            return(true);
        return(false);

/*
** JOKER-Zustand (Beliebiges Zeichen)
*/
      case CState::JOKER:

/*
** rfind wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::iterator siter=
                        states[state].nextstates.begin();
            siter!=states[state].nextstates.end();
            ++siter)
          if(r_find(*siter,iter+1,end))
            return(true);
        return(false);
    }
    return(false);
  }
```

Listing 73: Die Methode r_find (Forts.)

Die endgültigen Fassungen sollen wieder als Templates vorliegen:

```
template<typename Type>
size_t multiple_find(Type &s,
                     int nth,
                     size_t start=0,
                     bool complete=true) const;

template<typename Type>
bool r_find(int state, Type iter, Type &end) const;
```

Listing 74: multiple_find und r_find als Templates

Die Klasse `CAutomat` finden Sie auf der CD in den `CAutomat`-Dateien.

23 Wie findet man das n-te Vorkommen eines Musters in einem String von hinten?

Als Gegenstück zu `multiple_find` folgt hier die Methode `multiple_rfind`, die das n-te Vorkommen eines Musters von hinten nach vorne sucht.

multiple_rfind

Sie besitzt folgende Parameter:

- `s` ist der String, in dem gesucht wird.
- `sub` ist der Teilstring, der gesucht wird.
- `nth` definiert, das wievielte Vorkommen gesucht werden soll.
- `start` definiert die Position in `s`, ab der rückwärts gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Der Quellcode sieht folgendermaßen aus:

```
size_t multiple_rfind(const string &s,
                      int nth,
                      size_t start=string::npos,
                      bool complete=true) {

/*
```

Listing 75: Die Methode multiple_rfind

```
** Ungültige Werte?
*/
    if(nth<1||start<0)
      return(string::npos);

    if(start==string::npos)
      start=s.length()-1;

/*
** Bis zum n-ten Vorkommen durchlaufen
*/
    while(nth--) {
      string::const_iterator iter, end;

/*
** Position im String suchen, an der Muster erkannt wird
*/
      for(iter=s.begin()+start; iter!=s.end(); --iter,--start)
        if(r_find(0, iter,end))
          break;

/*
** Wurde Muster nicht gefunden?
*/
      if(iter==s.end())
        return(string::npos);

/*
** n-te Position erreicht?
*/
      if(!nth)
        return(start);

/*
** Zukünftige Suchposition außerhalb des Strings?
*/
      if(start<(static_cast<size_t>((complete)?distance(iter, end):1)))
        return(string::npos);

/*
** Complete==true?
```

Listing 75: Die Methode multiple_rfind (Forts.)

```
**  => Suchposition um eins verschieben
** complete==false?
**  => Suchposition um Länge des Teilstrings verschieben
*/
      start-=(complete)?distance(iter, end):1;
    }
    return(string::npos);
  }
```

Listing 75: Die Methode multiple_rfind (Forts.)

Und zum Schluss wieder die endgültige Fassung als Template:

```
template<typename Type>
size_t multiple_rfind(Type &s,
                      int nth,
                      size_t start=Type::npos,
                      bool complete=true) const;
```

Listing 76: multiple_rfind als Template

Die Klasse `CAutomat` finden Sie auf der CD in den `CAutomat`-Dateien.

24 Wie ermittelt man die Häufigkeit eines Teilstrings in einem String?

Mit angepasster Startposition muss so lange nach dem Teilstring gesucht werden, bis kein Vorkommen mehr gefunden wird.

count

Dazu wird eine Funktion `count` implementiert, die folgende Parameter besitzt:

- `s` ist der String, in dem gesucht wird.
- `sub` ist der Teilstring, der gesucht wird.
- `start` definiert die Position in `s`, ab der gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Die Bedeutung von `complete` ist identisch mit der in Rezept 20, »Wie findet man das n-te Vorkommen eines Teilstrings?«.

Die Funktion ist folgendermaßen aufgebaut:

```
int count(const string &s,
             const string &sub,
             size_t start=0,
             bool complete=true) {

/*
** Ungültiger Startwert?
*/
  if(start<0)
    return(0);

  int nth=0;

/*
** Solange Suchposition noch innerhalb des Strings
*/
  while(start<s.length()) {

  start=s.find(sub,start);

/*
** Teilstring gefunden?
**  => Anzahl um eins erhöhen
** Wenn nicht => Fertig.
*/
  if(start==string::npos)
    return(nth);
  else
    nth++;

/*
** Complete==true?
**  => Suchposition um eins verschieben
** complete==false?
**  => Suchposition um Länge des Teilstrings verschieben
*/
  start+=(complete)?sub.length():1;
  }
```

Listing 77: Die Methode count

```
  return(nth);
}
```

Listing 77: Die Methode count (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

25 Wie ermittelt man die Häufigkeit eines Musters in einem String?

Wir bauen dazu auf dem in Rezept 22 erweiterten Automaten `CAutomat` auf. Die dort implementierte `r_find`-Methode spielt beim Lösungsansatz eine wichtige Rolle.

count

Das grundlegende, in Rezept 24, »Wie ermittelt man die Häufigkeit eines Teilstrings in einem String?« erarbeitete Lösungsschema wird hier übernommen und in einer Methode `count` des Automaten `CAutomat` gekapselt, die die unten aufgeführten Parameter besitzt.

- `s` ist der String, in dem gesucht wird.
- `start` definiert die Position in `s`, ab der gesucht werden soll (optional).
- `complete` legt fest, ob nur nach kompletten Vorkommen von `sub` gesucht werden soll (optional).

Im weiteren Verlauf ist der Quellcode der Methode `count` aufgeführt.

```
  int count(const string &s,
                    size_t start=0,
                    bool complete=true) {

/*
** Ungültige Werte?
*/
    if(start<0)
      return(string::npos);

    int nth=0;

/*
** Den String komplett durchlaufen
```

Listing 78: Die Methode count

```
*/
    while(start<s.length()) {
      string::const_iterator iter, end;

/*
** Position im String suchen, an der Muster erkannt wird
*/
      for(iter=s.begin()+start; iter!=s.end(); ++iter,++start)
        if(r_find(0, iter,end))
          break;

/*
** Teilstring gefunden?
**  => Anzahl um eins erhöhen
** Wenn nicht => Fertig.
*/
      if(iter==s.end())
        return(nth);
      else
        nth++;

/*
** Complete==true?
**  => Suchposition um eins verschieben
** complete==false?
**  => Suchposition um Länge des Teilstrings verschieben
*/
      start+=(complete)?distance(iter, end):1;
    }
    return(nth);
  }
```

Listing 78: Die Methode count (Forts.)

Um alle Methoden des Automaten einheitlich zu halten, wollen wir auch `count` als Template definieren:

```
template<typename Type>
int count(Type &s,
```

Listing 79: Die Methode count als Template

```
          size_t start=0,
          bool complete=true) const;
```

Listing 79: Die Methode count als Template (Forts.)

Die Klasse `CAutomat` finden Sie auf der CD in den `CAutomat`-Dateien.

26 Wie kann in einem String nach einem Muster gesucht werden ohne die Groß-/Kleinschreibung zu berücksichtigen?

Grundlage der Lösung bildet die Klasse `CAutomat` aus Rezept 18.

In gewisser Weise bietet der Automat bereits eine Möglichkeit, die Groß- und Kleinschreibung zu ignorieren. Soll beispielsweise das Wort »test« ohne Beachtung der Groß- und Kleinschreibung gefunden werden, so wird einfach der set-Operator {} benutzt:

```
string s1("Dies ist ein Test im Glas");
CAutomat a("{Tt}{Ee}{Ss}{Tt}");
cout << "Gefunden an Position: " << a.find(s1) << endl;
```

Obwohl diese Vorgehensweise problemlos funktioniert, ist der dadurch erzeugte Automat viel komplexer und das Muster ist auch nicht gerade die intuitive Art, einen Suchbegriff zu formulieren.

findCI

Wir wollen daher auf die Prädikate aus Rezept 36 zurückgreifen und den Automaten um eine spezielle `find`-Methode erweitern. Wie die anderen Methoden des Automaten soll auch das neue `findCI` als Template implementiert werden:

```
template<typename Type>
size_t findCI(Type &s, size_t spos=0) const{
  for(Type::iterator iter=s.begin()+spos;
      iter!=s.end();
      ++iter,++spos)
    if(r_findCI(0, iter))
```

Listing 80: findCI von CAutomat

```
      return(spos);
  return(Type::npos);
}
```

Listing 80: findCI von CAutomat (Forts.)

r_findCI

Die Methode `r_findCI`, die den Automaten simuliert, ruft zur Überprüfung der Zeichen nicht mehr die STL-Funktion `find` auf, sondern verwendet `find_if` mit dem Prädikat `CCharCIPredicate`:

```
template<typename Type>
bool r_findCI(int state, Type iter) const{

/*
** Hat der Automat seinen Endzustand erreicht?
** => Muster wurde erkannt..
*/
  if(m_states[state].m_type==CState::END)
    return(true);

/*
** Ist das Ende der Zeichenkette erreicht, es müssen
** aber noch CHAR-Zustände des Automaten ausgewertet
** werden => Muster wurde  nicht erkannt.
*/
  if((*iter==0)&&(m_states[state].m_type!=CState::NOOP))
    return(false);

/*
** Status auswerten
*/
  switch(m_states[state].m_type) {

/*
** CHAR-Zustand
*/
    case CState::CHAR:

/*
```

Listing 81: Die Methode r_findCI von CAutomat

```
** Das aktuelle Zeichen der Zeichenkette wird ohne Berücksichtigung
** der Groß-/Kleinschreibung mit den Zeichen des Zustands
** verglichen. Stimmt es nicht überein, dann wurde das Muster
** nicht erkannt.
*/
      if(std::find_if(m_states[state].m_characters.begin(),
                      m_states[state].m_characters.end(),
                      CCharCIPredicate(*iter))==
         m_states[state].m_characters.end()) {
        return(false);
      } else {

/*
** Bei Übereinstimmung wird r_findCI
** rekursiv für jeden Folgezustand mit dem nächsten
** Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::const_iterator siter=
                        m_states[state].m_nextstates.begin();
            siter!=m_states[state].m_nextstates.end();
            ++siter)
          if(r_findCI(*siter,iter+1))
            return(true);
      }
        return(false);

/*
** NOOP- und START-Zustand
*/
    case CState::START:
    case CState::NOOP:

/*
** r_findCI wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
      for(vector<int>::const_iterator siter=
                      m_states[state].m_nextstates.begin();
          siter!=m_states[state].m_nextstates.end();
          ++siter)
        if(r_findCI(*siter,iter))
```

Listing 81: Die Methode r_findCI von CAutomat (Forts.)

```
          return(true);
      return(false);

/*
** JOKER-Zustand (beliebiges Zeichen)
*/
    case CState::JOKER:

/*
** r_findCI wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
      for(vector<int>::const_iterator siter=
                      m_states[state].m_nextstates.begin();
          siter!=m_states[state].m_nextstates.end();
          ++siter)
        if(r_findCI(*siter,iter+1))
          return(true);
      return(false);
  }
  return(false);
}
```

Listing 81: Die Methode r_findCI von CAutomat (Forts.)

matchCI

Um auch eine Möglichkeit der Mustererkennung ohne Berücksichtigung der Groß- und Kleinschreibung zu besitzen, wollen wir noch eine Methode `matchCI` hinzufügen:

```
template<typename Type>
bool matchCI(Type &s, size_t spos=0, size_t len=Type::npos) const{
  if((len!=Type::npos)&&((spos+len)>s.length()))
    return(false);          // Bereich überschritten
  return(r_matchCI(0, s.begin()+spos,
                 (len!=Type::npos)?s.begin()+spos+len:s.end()));
}
```

Listing 82: Die Methode matchCI von CAutomat

r_matchCI

Damit `matchCI` einwandfrei funktioniert, fehlt noch die Methode `r_matchCI`, die den Automaten simuliert.

```
template<typename Type>
bool r_matchCI(int state, Type iter, Type &end) const {

/*
** Hat die Zeichenkette ihr Ende erreicht?
*/
  if(iter==end) {

/*
** Hat der Automat seinen Endzustand erreicht?
** => Muster wurde erkannt..
*/
    if(m_states[state].m_type==CState::END)
      return(true);

/*
** Muss der Automat noch Zeichen auswerten?
** => Muster wurde  nicht erkannt..
*/
    if(m_states[state].m_type!=CState::NOOP)
      return(false);
  }

/*
** Status auswerten
*/
  switch(m_states[state].m_type) {

/*
** CHAR-Zustand
*/
    case CState::CHAR:

/*
** Das aktuelle Zeichen der Zeichenkette wird ohne Berücksichtigung
** der Groß-/Kleinschreibung mit den Zeichen des Zustands
** verglichen. Stimmt es nicht überein, dann wurde das Muster
```

Listing 83: Die Methode r_matchCI von CAutomat

```
** nicht erkannt.
*/
      if(std::find_if(m_states[state].m_characters.begin(),
                      m_states[state].m_characters.end(),
                      CCharCIPredicate(*iter))==
                      m_states[state].m_characters.end()) {
        return(false);
      } else {

/*
** Bei Übereinstimmung wird rmatch
** rekursiv für jeden Folgezustand mit dem nächsten
** Zeichen der Zeichenkette aufgerufen.
*/
        for(vector<int>::const_iterator siter=
                         m_states[state].m_nextstates.begin();
            siter!=m_states[state].m_nextstates.end();
            ++siter)
          if(r_matchCI(*siter,iter+1,end))
            return(true);
      }
        return(false);

/*
** NOOP- und START-Zustand
*/
    case CState::START:
    case CState::NOOP:

/*
** rmatch wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
      for(vector<int>::const_iterator siter=
                       m_states[state].m_nextstates.begin();
          siter!=m_states[state].m_nextstates.end();
          ++siter)
        if(r_matchCI(*siter,iter,end))
          return(true);
      return(false);
```

Listing 83: Die Methode r_matchCI von CAutomat (Forts.)

```
/*
** JOKER-Zustand (beliebiges Zeichen)
*/
    case CState::JOKER:

/*
** rmatch wird rekursiv für jeden Folgezustand
** mit dem aktuellen Zeichen der Zeichenkette aufgerufen.
*/
      for(vector<int>::const_iterator siter=
                      m_states[state].m_nextstates.begin();
          siter!=m_states[state].m_nextstates.end();
          ++siter)
        if(r_matchCI(*siter,iter+1, end))
          return(true);

      return(false);
  }
  return(false);
}
```

Listing 83: Die Methode r_matchCI von CAutomat (Forts.)

Die Klasse `CAutomat` finden Sie auf der CD in den `CAutomat`-Dateien.

27 Wie wird eine dezimale Zahl in einen hexadezimalen String umgewandelt?

Wir schreiben eine Funktion `valueToHex`, der ein numerischer Wert übergeben wird und die dann die hexadezimale Zahl (gegebenenfalls mit führenden Nullen) als String zurückgibt.

Sie besitzt folgende Parameter:

- `v` ist der umzuwandelnde numerische Wert.
- `s` ist die Breite der zu erzeugenden hexadezimalen Zahl (standardmäßig auf 2).

Die Funktion ist folgendermaßen aufgebaut:

```
template<typename Type>
string valueToHex(Type v, int s=2) {
```

Listing 84: Die Funktion valueToHex

```
  ostringstream ostr;
  ostr << setfill('0') << setw(s) << hex << v;
  return(ostr.str());
}
```

Listing 84: Die Funktion valueToHex (Forts.)

Die Zahl wird in einen Stringstream geschrieben. Dabei wird mit den entsprechenden Manipulatoren dafür gesorgt, dass die Zahl im hexadezimalen Format ausgegeben wird und sie die entsprechende Anzahl führender Nullen besitzt.

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

28 Wie wird ein hexadezimaler String in einen numerischen Wert umgewandelt?

Dieses Problem ist denkbar einfach zu lösen. Wir greifen auf die Funktion `strtol` zurück, die einen C-String und die gewünschte Basis übergeben bekommt und dann den numerischen Wert in Form eines `long`-Wertes zurückliefert.

Wir kapseln diese Funktionalität in der Funktion `hexToValue`:

```
long hexToValue(string s) {
  return(strtol(s.c_str(),0,16));
}
```

Listing 85: Die Funktion hexToValue

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

29 Wie wird eine Ganzzahl in eine römische Zahl umgewandelt?

Wir schreiben dazu eine einfache Funktion `toRoman`. Der boolesche Wert `upcase` ist mit dem Standardwert `true` versehen und entscheidet, ob die römische Zahl in Großbuchstaben (`upcase=true`) oder Kleinbuchstaben (`upcase=true`) dargestellt werden soll.

```
string toRoman(unsigned int val, bool upcase) {

/*
** Statische Umwandlungstabellen setzen
*/
  static unsigned int intValues[]={1000,900,500,400,
                                   100,90,50,40,
                                   10,9,5,4,1};
  static char *romanValues[]={"M","CM","D","CD",
                              "C","XC","L","XL",
                              "X","IX","V","IV","I"};

/*
** Wenn 0, dann leeren String zurückgeben
*/
  if(val==0)
    return("");
  string str;
  int pos=0;

/*
** Schritt für Schritt Werte aus intValues abziehen
** und Zeichenketten aus romanValues zum Ergebnis hinzufügen
*/
  while(val) {
    if(val>=intValues[pos]) {
      str+=romanValues[pos];
      val-=intValues[pos];
    }
    else
      pos++;
  }

/*
** Sollen Kleinbuchstaben erzeugt werden?
** => toLowerString aufrufen
*/
  if(upcase)
    return(str);
  else
    return(toLowerString(str));
}
```

Listing 86: Die Funktion toRoman

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

Mit dieser Funktion kann auch die Frage auf ein kleines Rätsel beantwortet werden: Welches Jahr der ersten 2000 Jahre nach Christi braucht in der römischen Schreibweise am meisten Zeichen? Auflösung in der Fußnote[1].

30 Wie kann eine römische Zahl in eine Ganzzahl umgewandelt werden?

Diese Funktion ist gewissermaßen die Umkehrfunktion zu `toRoman` aus dem vorigen Rezept. Nennen wir sie `romanToUnsignedInt`.

```
unsigned int romanToUnsignedInt(const string &str) {

/*
** Statische Umwandlungstabellen setzen
*/
  static unsigned int intValues[]={900,400,90,40,9,4,
                                   1000,500,100,50,10,5,1};
  static char *romanValues[]={"CM","CD","XC","XL","IX","IV",
                              "M","D","C","L","X","V","I"};

/*
** Römische Zahl in Großbuchstaben umwandeln
*/
  string roman=toUpperString(str);
  size_t pos=0;
  unsigned int value=0;

/*
** Schritt für Schritt nach Stücken römischer Zahlen im String
** suchen und den Gegenwert zu value hinzuaddieren
*/
  while(pos<roman.size()) {
    int x;
    for(x=0; x<13; x++)
      if(roman.substr(pos,strlen(romanValues[x]))==romanValues[x]) {
```

Listing 87: Die Funktion romanToUnsignedInt

1. Es ist das Jahr 1888. Die römische Darstellung hat mit MDCCCLXXXVIII genau 13 Zeichen, so viel wie kein anderes Jahr in diesem Rezept.

```
        pos+=strlen(romanValues[x]);
        value+=intValues[x];
        break;
      }

/*
** Wurde kein Stück gefunden?
** => keine gültige römische Zahl
*/
    if(x==13)
      throw invalid_argument("romanToUnsignedInt()");
  }
  return(value);
```

Listing 87: Die Funktion romanToUnsignedInt (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

31 Wie kann überprüft werden, ob ein String nur aus Zahlen besteht?

Wir schreiben dazu eine Funktion `isNumericString`, die einen String übergeben bekommt und über ihren booleschen Rückgabewert mitteilt, ob der String nur aus Zahlen besteht oder nicht.

```
bool isNumericString(const string &s) {
  for(string::const_iterator iter=s.begin(); iter!=s.end(); ++iter)
    if(!isdigit(*iter))
      return(false);
  return(true);
}
```

Listing 88: Die Funktion isNumericString

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

32 Wie werden deutsche Umlaute bei ctype-Funktionen korrekt behandelt?

Vermutlich ist es Ihnen bereits aufgefallen. Die Funktionen aus `ctype` arbeiten bei deutschen Umlauten oder dem ß nicht korrekt. Schlimmstenfalls bricht die Funktion im Debug-Modus wegen eines `assert` ab.

Dieses Problem kann behoben werden, indem wir auf die `locale`- und `facet`-Objekte der Standardbibliothek zurückgreifen. Um diese Objekte nicht bei jedem Funktionsaufruf neu anlegen oder jeder Funktion ein eigenes statisches Objekt mitgeben zu müssen, legen wir die benötigte Facette als globales Objekt an.

Wir kapseln sie aber innerhalb unseres eigenen `Codebook`-Namensbereichs nochmals in einem eigenständigen Namensbereich `CBLocale`. Auf diese Weise sollte die Wahrscheinlichkeit einer Kollision mit anderen globalen Objekten nahezu ausgeräumt sein:

```
namespace CBLocale {
  const locale Locale("German_Germany");
  const ctype<char> &Facet=use_facet<ctype<char> >(Locale);
};
```

Listing 89: Die Facette für die eigenen ctype-Funktionen

Nun implementieren wir in unserem Namensbereich `Codebook` die eigenen `ctype`-Funktionen:

```
using namespace CBLocale;

char tolower(char c) {
  return(Facet.tolower(c));
}

//*********

char toupper(char c) {
  return(Facet.toupper(c));
}

//*********
```

Listing 90: Die eigenen ctype-Funktionen

```
bool isalnum(char c) {
  return(Facet.is(ctype_base::alnum,c));
}

//*********

bool isalpha(char c) {
  return(Facet.is(ctype_base::alpha,c));
}

//*********

bool iscntrl(char c) {
  return(Facet.is(ctype_base::cntrl,c));
}

//*********

bool isdigit(char c) {
  return(Facet.is(ctype_base::digit,c));
}

//*********

bool isgraph(char c) {
  return(Facet.is(ctype_base::graph,c));
}

//*********

bool islower(char c) {
  return(Facet.is(ctype_base::lower,c));
}

//*********

bool isprint(char c) {
  return(Facet.is(ctype_base::print,c));
}

//*********
```

Listing 90: Die eigenen ctype-Funktionen (Forts.)

```
bool isupper(char c) {
  return(Facet.is(ctype_base::upper,c));
}

//*********

bool isxdigit(char c) {
  return(Facet.is(ctype_base::xdigit,c));
}
```

Listing 90: Die eigenen ctype-Funktionen (Forts.)

Die Funktion finden Sie auf der CD in den `StringFunctions`-Dateien.

Es muss aber auch ganz klar hervorgehoben werden, dass die lokalisierten Varianten der `ctype`-Funktionen erheblich mehr Aufwand betreiben müssen und damit auch um einiges langsamer sind.

Eine Implementierung auf dem Visual C++ Compiler von Microsoft benötigte mehr als sechs Mal so viel Zeit wie die Funktionen aus `ctype`.

33 Wie wird das erste Element gesucht, das nicht in einem anderen Container enthalten ist?

Wir benötigen dazu das Gegenstück zum STL-Algorithmus `find_first_of`.

find_first_not_of

```
template<class Forward1, class Forward2> inline
Forward1 find_first_not_of(Forward1 first1, Forward1 last1,
                           Forward2 first2, Forward2 last2) {
  for (; first1 != last1; ++first1) {
    for (Forward2 mid2 = first2; mid2 != last2; ++mid2)
      if (*first1 == *mid2)
        break;
    if(mid2==last2)
      return(first1);
  }
  return (first1);
}
```

Listing 91: Das Template find_first_not_of

Die Funktion finden Sie auf der CD in der `STLFunctions.h`-Datei.

Das Problem könnte auch gelöst werden, indem das Template `find_first_of` mit dem Prädikat `not_equal_to` eingesetzt würde.

34 Wie kann der Index-Operator bei Vektoren gesichert werden?

Der wahlfreie Zugriff auf Elemente eines STL-Vektors ist im Normalfall auf zwei Arten möglich: entweder mit dem Index-Operator oder mit der Methode `at`.

Der einzige Unterschied liegt darin, dass die Methode `at` den angegebenen Index auf Gültigkeit prüft und gegebenenfalls eine `out_of_range`-Ausnahme wirft, wohin-

gegen der Index-Operator ungebremst den Index verwendet und schlimmstenfalls Speicherbereiche außerhalb des Vektor-Speichers anspricht. Mit meist unangenehmen Folgen.

Es wäre doch schön, wenn der eleganter einzusetzende Index-Operator ebenfalls den Index prüfen und bei Bedarf eine Ausnahme werfen würde.

Wir implementieren dazu eine Klasse `CSaveVector`, die öffentlich von der STL-Klasse `vector` abgeleitet wird.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CSaveVector : public std::vector<Type, Allocator> {

public:
  typedef std::vector<Type, Allocator> base_type;
  typedef CSaveVector<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 92: Die Klassendefinition von CSaveVector

Konstruktoren

Durch die öffentliche Vererbung stehen in unserer Klasse alle Methoden von `vector` zur Verfügung – bis auf die Konstruktoren. Wir müssen für unsere Klasse eigene Konstruktoren implementieren und über ihre Elementinitialisierungslisten dann die Basisklassen-Konstruktoren aufrufen.

```
explicit CSaveVector(const Allocator &a=Allocator())
  : base_type(a)
{}

CSaveVector(const my_type &v)
  : base_type(v)
{}

CSaveVector(size_type n,
```

Listing 93: Die Konstruktoren von CSaveVector

```
            const Type &obj=Type(),
            const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}

template<typename Input>
CSaveVector(Input begin, Input end, const Allocator &a=Allocator())
  : base_type(begin, end ,a)
{}
```

Listing 93: Die Konstruktoren von CSaveVector (Forts.)

Index-Operatoren

Die Index-Operatoren schließlich sind diejenigen, wegen denen wir den ganzen Aufwand überhaupt betreiben. Um ihren Zugriff zu sichern, benutzen wir für den Zugriff die `at`-Methode.

```
reference operator[](size_type offset) {
  return(at(offset));
}

const_reference operator[](size_type offset) const {
  return(at(offset));
}
```

Listing 94: Die Index-Operatoren von CSaveVector

Die Klasse `CSaveVector` finden Sie auf der CD in der `CSaveVector.h`-Datei.

35 Wie kann der Index-Operator bei Deques gesichert werden?

Motivation dieser Frage und Vorgehensweise der Lösung sind nahezu identisch mit der Klasse `CSaveVector` aus Rezept 34.

Wir implementieren dazu eine Klasse `CSaveDeque`, die öffentlich von der STL-Klasse `deque` abgeleitet wird.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CSaveDeque : public std::deque<Type, Allocator> {

public:
  typedef std::deque<Type, Allocator> base_type;
  typedef CSaveDeque<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 95: Die Klassendefinition von CSaveDeque

Konstruktoren

Die Basisklassen-Konstruktoren müssen über die Elementinitialisierungslisten der eigenen Konstruktoren aufgerufen werden.

```
explicit CSaveDeque(const Allocator &a=Allocator())
  : base_type(a)
{}

CSaveDeque(const my_type &v)
  : base_type(v)
{}

CSaveDeque(size_type n,
           const Type &obj=Type(),
           const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}

template<typename Input>
CSaveDeque(Input begin, Input end, const Allocator &a=Allocator())
  : base_type(begin, end ,a)
{}
```

Listing 96: Die Konstruktoren von CSaveDeque

Index-Operatoren

Zum Schluss fehlen noch die Index-Operatoren, die ihre Arbeit mit Hilfe der `at`-Methode verrichten.

```
reference operator[](size_type offset) {
  return(at(offset));
}

const_reference operator[](size_type offset) const {
  return(at(offset));
}
```

Listing 97: Die Index-Operatoren von CSaveDeque

Die Klasse `CSaveDeque` finden Sie auf der CD in der *CSaveDeque.h*-Datei.

36 Wie können die find-Algorithmen ohne Beachtung der Groß-/Kleinschreibung eingesetzt werden?

Will man mit Hilfe von `find` ein Zeichen in einem Container finden, dann sieht das im Normalfall so aus[1]:

```
string s1("Dies ist ein Test im Glas");
string::iterator i=find(s1.begin(), s1.end(), 'T');
cout << "Gefunden: " << *i << endl;
```

Das obere Codestück findet den Buchstaben »T« im Wort »Test« und nicht das »t« aus »ist«, weil zwischen Groß- und Kleinschreibung unterschieden wird.

CCharCIPredicate

Wie ist nun aber vorzugehen, wenn es keine Rolle spielen soll, ob nun ein »t« oder ein »T« gefunden wird?

1. In diesem konkreten Fall hätte auch die `find`-Methode von `string` zum Einsatz kommen können. Aber wir wollen die Betrachtung allgemeingültiger halten.

Dazu implementieren wir uns das Prädikat `CCharCIPredicate`, welches die Unterscheidung zwischen Groß- und Kleinschreibung für uns ignoriert. Um konform mit den Prädikaten der STL zu gehen, wird es von `unary_function` angeleitet:

```
class CCharCIPredicate : public std::unary_function<char, bool> {
private:
  char m_char;
public:
  CCharCIPredicate(char c)
    : m_char(tolower(c))
  {}
  bool operator()(char o) {
    return(m_char==tolower(o));
  }
};
```

Listing 98: Das Prädikat CCharCIPredicate

Das Prädikat kann nun über `find_if` eingesetzt werden:

```
  string s1("Dies ist ein Test im Glas");
  string::iterator i=find_if(s1.begin(),
                             s1.end(),
                             CCharCIPredicate('T'));
  cout << "Gefunden: " << *i << endl;
```

Nun wird das kleine »t« aus »ist« gefunden, weil das Prädikat die Groß-/Kleinschreibung ignoriert.

CCharCIBinPredicate

Die gleiche Funktionalität wollen wir nun auch beim Vergleich von Sequenzen zur Verfügung stellen.

Beim Vergleich von Sequenzen brauchen wir ein binäres Prädikat. Es wird aus Kompatibilitätsgründen von `binary_function` abgeleitet:

```
class CCharCIBinPredicate
: public std::binary_function<char, char, bool> {
```

Listing 99: Das binäre Prädikat CCharCIBinPredicate

```
public:
  bool operator()(char o1, char o2) {
    return(tolower(o1)==tolower(o2));
  }
};
```

Listing 99: Das binäre Prädikat CCharCIBinPredicate (Forts.)

Nun lassen sich auch Sequenzen ohne Berücksichtigung der Groß- und Kleinschreibung vergleichen:

```
  string s1("Dies ist ein Test im Glas");
  string s2("tEsT");
  string::iterator i=search(s1.begin(),
                            s1.end(),
                            s2.begin(),
                            s2.end(),
                            CCharCIBinPredicate());
  cout << "Gefunden an Position: " <<
          distance(s1.begin(),i) << endl;
```

Es wird Position 13 ausgegeben.

Die beiden Prädikate finden Sie auf der CD in der *CCharCIPredicate.h*-Datei.

37 Wie kann eine Matrix implementiert werden?

Es gibt viele mögliche Ansätze, eine Matrix zu implementieren. Grundsätzlich sollte eine vernünftige Implementierung auf die Container der STL zurückgreifen, um das Rad nicht neu erfinden zu müssen.

Wenn man versucht, in Begriffen der Objektorientierten Programmierung zu denken, könnte man auf den Ansatz kommen, dass eine Matrix für einen bestimmten Datentyp nichts anderes ist als ein Vektor von Vektoren des Datentyps. Also in etwa wie folgt:

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix : public std::vector<std::vector<Type> > {
};
```

Dieser Ansatz hat den Vorteil, dass die übliche Index-Schreibweise, wie sie in C++ bei mehrdimensionalen Feldern üblich ist (z.B. `f[2][3]`), verwendet werden kann.

So elegant dieser Ansatz auf den ersten Blick erscheinen mag, hat er jedoch den dramatischen Nachteil, dass die interne Struktur der Matrix nicht gekapselt ist und somit von außen problemlos in einen inkonsistenten Zustand gebracht werden kann. Ein Beispiel:

```
CMatrix2<int> m(5,5);
m[0].clear();
```

Wenn wir davon ausgehen, dass die Matrix mit einem Konstruktor ausgestattet wurde, dem die Größe der Matrix übergeben wird, dann haben wir mit der ersten Zeile eine 5*5-Matrix erzeugt.

Nun ist es möglich, lediglich einen Index zu verwenden und damit alle Methoden der `vector`-Klasse anzusprechen. In diesem Fall wurde der Vektor der ersten Zeile (oder Spalte, je nach interner Implementierung) gelöscht. Die Matrix ist damit nicht mehr funktionstüchtig.

Es sollte also ein Ansatz gewählt werden, bei dem der Benutzer nicht in die interne Struktur eingreifen kann. Andererseits sollte aber der Vorteil der Index-Schreibweise beibehalten werden.

CMatrixVector

Wir werden daher speziell für die Matrix eine Vektor-Klasse erstellen, die nur die Implementierung des üblichen STL-Vektors übernimmt, die Schnittstelle aber bis auf die für den Benutzer wesentlichen Teile (Index-Operator) kapselt.

Klassendefinition

Die Klasse `CMatrixVector` wird später von der Klasse `CMatrix` eingesetzt. Damit die Matrix selbst vollen Zugriff auf die gekapselten Methoden besitzt, wird sie als Freund von `CMatrixVector` deklariert. Dazu muss vorher die spätere Matrix-Klasse deklariert werden.

Wir leiten `CMatrixVector` von der Klasse `CSaveVector` aus Rezept 34 ab, damit ein sicherer Zugriff über den Index-Operator gewährleistet ist. Bei Bedarf kann natürlich auch der übliche STL-Vektor genommen werden.

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix;

template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrixVector : private CSaveVector<Type, Allocator> {
  friend class CMatrix;
public:
  typedef CSaveVector<Type, Allocator> base_type;
  typedef CMatrixVector<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
  typedef base_type::const_reference const_reference;
};
```

Listing 100: Die Klassendefinition von CMatrixVector

Konstruktoren

Von den Konstruktoren der Basisklasse werden lediglich zwei »durchgeschliffen«.

```
explicit CMatrixVector(const Allocator &a=Allocator())
  : base_type(a)
{}

//*********

CMatrixVector(size_type n,
              const Type &obj=Type(),
              const Allocator &a=Allocator())
  : base_type(n,obj,a)
{}
```

Listing 101: Die Konstruktoren von CMatrixVector

Index-Operatoren

Damit der Index-Operator für Außenstehende einsetzbar ist, muss eine öffentliche Variante existieren.

```
reference operator[](size_type idx) {
  return(base_type::operator[](idx));
```

Listing 102: Die Index-Operatoren von CMatrixVector

```
}

//*********

const_reference operator[](size_type idx) const {
  return(base_type::operator[](idx));
}
```

Listing 102: Die Index-Operatoren von CMatrixVector (Forts.)

CMatrix

Die Klasse `CMatrix` besitzt als wichtigstes Attribut einen `CMatrixVector` vom Typ `CMatrixVector<Type>`. Dadurch können wir später die gewohnte Index-Schreibweise einsetzen, haben aber die restlichen Methoden des Vektors in `CMatrixVector` gekapselt und nur für `CMatrix` zugänglich gemacht.

Zusätzlich definieren wir noch die Attribute `m_rows` und `m_columns` für die Dimensionen der Matrix.

Die Dimensionen der Matrix lassen sich über Zugriffsmethoden ermitteln. Darüber hinaus wird noch die Methode `size` implementiert, die die Anzahl der Elemente in der Matrix liefert.

Klassendefinition

```
template<typename Type, typename Allocator=std::allocator<Type> >
class CMatrix {
public:
  typedef CMatrix<Type, Allocator> my_type;
  typedef std::size_t size_type;
  typedef Type& reference;
  typedef const Type& const_reference;

private:
  CMatrixVector<CMatrixVector<Type, Allocator>, Allocator>
    m_elements;
  size_type m_rows;
  size_type m_columns;
```

Listing 103: Klassendefinition und Zugriffsmethoden von CMatrix

```
public:
  size_type getRows() const {
    return(m_rows);
  }

//*********

  size_type getColumns() const {
    return(m_columns);
  }

//*********

  size_type size() const {
    return(m_rows*m_columns);
  }
};
```

Listing 103: Klassendefinition und Zugriffsmethoden von CMatrix (Forts.)

Konstruktoren

Die Matrix erhält einen Standard-Konstruktor, einen Kopier-Konstruktor und einen Konstruktor, mit dem eine Matrix dimensioniert und optional alle Werte mit einem Wert initialisiert werden können.

```
CMatrix()
: m_rows(0), m_columns(0)
{}

//*********

CMatrix(const CMatrix &m)
: m_elements(m.m_elements), m_rows(m.m_rows), m_columns(m.m_columns)
{ }

//*********

CMatrix(size_type rows, size_type columns, const Type &obj=Type()) {
  resize(rows,columns,obj);
}
```

Listing 104: Einige Konstruktoren von CMatrix

Zusätzlich wird ein Konstruktor implementiert, der die Matrix aus einem Teil einer anderen Matrix erzeugt. Dazu werden die Position innerhalb der Matrix und die gewünschte Größe angegeben:

```
CMatrix(const CMatrix &m,
        size_type rbegin, size_type cbegin,
        size_type rows, size_type columns) {
  resize(rows,columns);
  for(size_type r=0; r<rows; ++r)
    for(size_type c=0; c<columns; ++c)
      m_elements[r][c]=m.m_elements[r+rbegin][c+cbegin];
}
```

Listing 105: Ein Konstruktor für Bereiche einer Matrix

Ferner wird noch ein Konstruktor hinzugefügt, der über zwei Iteratoren eine einspaltige bzw. einzeilige Matrix erzeugt:

```
template<typename IType>
CMatrix(bool column, IType beg, IType end) {
  if(column) {
    resize(distance(beg,end),1);
    copy(beg,end,column_begin());
  }
  else {
    resize(1, distance(beg,end));
    copy(beg,end,row_begin());
  }
}
```

Listing 106: Ein Konstruktor für einzeilige/-spaltige Matrizen

Zuweisungsoperator

```
CMatrix &operator=(const CMatrix &m) {
  if(this!=&m) {
    m_elements=m.m_elements;
    m_rows=m.m_rows;
```

Listing 107: Der Zuweisungsoperator von CMatrix

```
    m_columns=m.m_columns;
  }
  return(*this);
}
```

Listing 107: Der Zuweisungsoperator von CMatrix (Forts.)

Index-Operatoren

Die Index-Operatoren von `CMatrix` rufen die Index-Operatoren von `CMatrixVector` auf.

```
CMatrixVector<Type, Allocator> &operator[](size_type idx) {
  return(m_elements[idx]);
}

//*********

const CMatrixVector<Type, Allocator>
&operator[](size_type idx) const {
  return(m_elements[idx]);
}
```

Listing 108: Die Index-Operatoren von CMatrix

Aufruf-Operatoren

Wir wollen die Klasse auch mit Aufruf-Operatoren versehen, um eine alternative Schreibweise für den Elementzugriff zu ermöglichen. Obwohl der Aufruf-Operator bei der gewählten internen Struktur keine Geschwindigkeitsvorteile bringt, kann er insbesondere bei Implementierungen für dünn besetzte Matrizen von Vorteil sein.

```
Type &operator()(size_type r, size_type c) {
  return(m_elements.at(r).at(c));
}

//*********

const Type &operator()(size_type r, size_type c) const {
```

Listing 109: Die Aufruf-Operatoren von CMatrix

```
  return(m_elements.at(r).at(c));
}
```

Listing 109: Die Aufruf-Operatoren von CMatrix (Forts.)

resize

Mit der öffentlichen Methode `resize` kann die Größe der Matrix angepasst werden. Einige der Konstruktoren machen von ihr Gebrauch.

```
void resize(size_type rows,
            size_type columns,
            const Type &obj=Type()) {

/*
** Größe des Zeilenvektors anpassen
*/
  m_elements.resize(rows, CMatrixVector<Type,
                    Allocator>(columns,obj) );

/*
** Größe der Spaltenvektoren anpassen
*/
  for(size_type r=0; r<rows; ++r)
    m_elements[r].resize(columns,obj);
  m_columns=columns;
  m_rows=rows;
}
```

Listing 110: Die Methode resize von CMatrix

output

Damit die Matrix einigermaßen strukturiert ausgegeben werden kann, fügen wir die Methode `output` hinzu, die eine maximale Elementbreite für die Ausgabe übergeben bekommt und die Matrix als String zurückliefert.

```
std::string output(int width) const{
  ostringstream os;
  for(size_type y=0; y<m_rows; ++y) {
```

Listing 111: Die Methode output von CMatrix

```
    for(size_type x=0; x<m_columns; ++x)
      os << std::setw(width) << (*this)[y][x] << " ";
    os << "\n";
  }
  return(os.str());
}
```

Listing 111: Die Methode output von CMatrix (Forts.)

Stream-Operatoren

Damit ein Objekt vom Typ `CMatrix` auch mit den binären Strömen aus Rezept 106 zusammenarbeiten kann, überladen wir hier noch die Stream-Operatoren:

```
friend CBinaryOStream &operator<<(CBinaryOStream &os,
                                  const CMatrix &m){
  os << m.m_rows;
  os << m.m_columns;
  const_row_iterator iter=m.row_begin();
  while(iter!=m.row_end())
    os << *(iter++);
  return(os);
}

friend CBinaryIStream &operator>>(CBinaryIStream &is, CMatrix &m){
  is >> m.m_rows;
  is >> m.m_columns;
  m.resize(m.m_rows,m.m_columns);
  row_iterator iter=m.row_begin();
  while(iter!=m.row_end())
    is >> *(iter++);
  return(is);
}
```

Listing 112: Die Stream-Operatoren von CMatrix

row_iterator

Damit wir mit der Matrix-Klasse elegant arbeiten können, ergänzen wir sie um Iteratoren. Die Frage ist nur, wie ein Iterator über die einzelnen Elemente der Matrix iterieren sollte.

In manchen Fällen macht es Sinn, die Matrix zeilenweise zu durchlaufen, in bestimmten Situationen kann aber auch eine spaltenweise Iteration vonnöten sein.

Wir werden deswegen zwei verschiedene Iteratoren programmieren, einen für die zeilenweise Iteration und einen für das spaltenweise Durchlaufen.

Beginnen wir mit `row_iterator`, dem Iterator für die zeilenweise Iteration.

Als Attribute benötigen wir einen Zeiger auf die dazugehörige Matrix und die Koordinaten des aktuellen Elementes.

Klassendefinition

```
class row_iterator;
friend class row_iterator;
class row_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  CMatrix *m_matrix;
};
```

Listing 113: Die Klassendefinition von row_iterator

Konstruktoren

Es gibt einen Standard-Konstruktor und einen Konstruktor, der das Iterator-Objekt mit den Koordinaten in einer bestimmten Matrix verknüpft.

```
row_iterator()
  :m_matrix(0), m_r(0), m_c(0)
{}

//*********

row_iterator(CMatrix &matrix, size_type r, size_type c)
  : m_matrix(&matrix), m_r(r), m_c(c)
{}
```

Listing 114: Die Konstruktoren von row_iterator

Inkrement-/Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren sind das Herzstück eines bidirektionalen Iterators.

Die Post-Operatoren greifen auf die Funktionalität der Prä-Operatoren zurück.

```
row_iterator &operator++() {

/*
** Ende-Position erreicht?
** Ende-Position beibehalten
*/
  if(m_r==m_matrix->m_rows)
    return(*this);

/*
** Nächstes Element der Zeile
*/
  ++m_c;

/*
** Zeile zu Ende?
** => Auf erstes Element der nächsten Zeile setzen
*/
  if(m_c==m_matrix->m_columns) {
    m_c=0;
    ++m_r;
  }
  return(*this);
}

//*********

row_iterator &operator--() {

/*
** Anfangs-Position erreicht?
** Anfangs-Position beibehalten
*/
  if((m_c==0)&&(m_r==0))
    return(*this);
```

Listing 115: Die Inkrement-/Dekrement-Operatoren von row_iterator

```
/*
** Aktuelles Element erstes Element der Zeile?
** => Auf letztes Element der vorhergehenden Zeile setzen
*/
  if(m_c==0) {
    m_c=m_matrix->m_columns-1;
    --m_r;
  }

/*
** Andernfalls auf vorhergehendes Element der Zeile setzen
*/
  else
    --m_c;
  return(*this);
}

//*********

row_iterator operator++(int) {
  row_iterator tmp=*this;
  ++(*this);
  return(tmp);
}

//*********

row_iterator operator--(int) {
  row_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 115: Die Inkrement-/Dekrement-Operatoren von row_iterator (Forts.)

Zugriffsoperatoren

Zum Schluss fehlen noch die Operatoren, mit denen das mit dem Iterator verknüpfte Element der Matrix angesprochen werden kann.

```
Type &operator*() {
  return((*m_matrix)[m_r][m_c]);
}

//*********

Type *operator->() {
  return(&(*m_matrix)[m_r][m_c]);
}
```

Listing 116: Die Zugriffsoperatoren von row_iterator

Vergleichsoperatoren

Bidirektionale Iteratoren müssen die Vergleichsoperationen == und != unterstützen:

```
bool operator==(row_iterator &i) const {
  return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
}

//*********

bool operator!=(row_iterator &i) const {
  return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
}
```

Listing 117: Die Vergleichsoperatoren von row_iterator

column_iterator

Der `column_iterator` ist so konzipiert, dass er die Matrix spaltenweise durchläuft.

Klassendefinition

```
class column_iterator;
friend class column_iterator;
class column_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
```

Listing 118: Die Klassendefinition von column_iterator

```
  size_type m_c;
  CMatrix *m_matrix;
};
```

Listing 118: Die Klassendefinition von column_iterator (Forts.)

Konstruktoren

Die Konstruktoren werden analog zu denen von `row_iterator` implementiert.

```
column_iterator()
  :m_matrix(0), m_r(0), m_c(0)
{}

//*********

column_iterator(CMatrix &matrix, size_type r, size_type c)
  : m_matrix(&matrix), m_r(r), m_c(c)
{}
```

Listing 119: Die Konstruktoren von column_iterator

Inkrement-/Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren von `column_iterator` sorgen dafür, dass der Iterator spaltenweise über die Matrix wandert.

```
column_iterator &operator++() {

/*
** Ende-Position erreicht?
** Ende-Position beibehalten
*/
  if(m_c==m_matrix->m_columns)
    return(*this);

/*
** Nächstes Element der Spalte
*/
  ++m_r;
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator

```
/*
** Spalte zu Ende?
** => Auf erstes Element der nächsten Spalte setzen
*/
  if(m_r==m_matrix->m_rows) {
    m_r=0;
    ++m_c;
  }
  return(*this);
}

//*********

column_iterator &operator--() {

/*
** Anfangs-Position erreicht?
** Anfangs-Position beibehalten
*/
  if((m_c==0)&&(m_r==0))
    return(*this);

/*
** Ist aktuelles Element erstes Element der Spalte?
** => Auf letztes Element der vorhergehenden Spalte setzen
*/
  if(m_r==0) {
    m_r=m_matrix->m_rows-1;
    --m_c;
  }

/*
** Andernfalls auf vorhergehendes Element der Spalte setzen
*/
  else
    --m_r;
  return(*this);
}

//*********
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator (Forts.)

```
column_iterator operator++(int) {
  column_iterator tmp=*this;
  ++(*this);
  return(tmp);
}

//*********

column_iterator operator--(int) {
  column_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 120: Die Inkrement-/Dekrement-Operatoren von column_iterator (Forts.)

Zugriffsoperatoren

```
Type &operator*() {
  return((*m_matrix)[m_r][m_c]);
}

//*********

Type *operator->() {
  return(&(*m_matrix)[m_r][m_c]);
}
```

Listing 121: Die Zugriffsoperatoren von column_iterator

Vergleichsoperatoren

```
bool operator==(column_iterator &i) const {
  return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
}

//*********
```

Listing 122: Die Vergleichsoperatoren von column_iterator

```
bool operator!=(column_iterator &i) const {
  return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
}
```

Listing 122: Die Vergleichsoperatoren von column_iterator (Forts.)

const-Iteratoren

Damit die Iteratoren auch mit konstanten Matrizen arbeiten können, benötigen wir `row_iterator` und `column_iterator` zusätzlich für konstante Objekte.

Wir nennen sie `const_row_iterator` und `const_column_iterator`. Der Vollständigkeit halber sind sie im Folgenden aufgeführt.

```
//*****************************************************************
// const_row_iterator
//*****************************************************************

class const_row_iterator;
friend class const_row_iterator;
class const_row_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  const CMatrix *m_matrix;
public:
  const_row_iterator()
    :m_matrix(0), m_r(0), m_c(0)
  {}

//**********

  const_row_iterator(const CMatrix &matrix,
                     size_type r,
                     size_type c)
    : m_matrix(&matrix), m_r(r), m_c(c)
  {}

//**********
```

Listing 123: Die const-Iteratoren von CMatrix

```
  const_row_iterator &operator++() {
    if(m_r==m_matrix->m_rows)
      return(*this);
    ++m_c;
    if(m_c==m_matrix->m_columns) {
      m_c=0;
      ++m_r;
    }
    return(*this);
  }

//*********

  const_row_iterator &operator--() {
    if((m_c==0)&&(m_r==0))
      return(*this);
    if(m_c==0) {
      m_c=m_matrix->m_columns-1;
      --m_r;
    }
    else
      --m_c;
    return(*this);
  }

//*********

  const_row_iterator operator++(int) {
    const_row_iterator tmp=*this;
    ++(*this);
    return(tmp);
  }

//*********

  const_row_iterator operator--(int) {
    const_row_iterator tmp=*this;
    --(*this);
    return(tmp);
  }
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
//*********

  const Type &operator*() {
    return((*m_matrix)[m_r][m_c]);
  }

//*********

  const Type *operator->() {
    return(&(*m_matrix)[m_r][m_c]);
  }

//*********

  bool operator==(const_row_iterator &i) const {
    return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
  }

//*********

  bool operator!=(const_row_iterator &i) const {
    return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
  }
};


//*****************************************************************
// const_column_iterator
//*****************************************************************

class const_column_iterator;
friend class const_column_iterator;
class const_column_iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  size_type m_r;
  size_type m_c;
  const CMatrix *m_matrix;
public:
  const_column_iterator()
    :m_matrix(0), m_r(0), m_c(0)
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
  {}

//*********

  const_column_iterator(const CMatrix &matrix,
                        size_type r,
                        size_type c)
    : m_matrix(&matrix), m_r(r), m_c(c)
  {}

//*********

  const_column_iterator &operator++() {
    if(m_c==m_matrix->m_columns)
      return(*this);
    ++m_r;
    if(m_r==m_matrix->m_rows) {
      m_r=0;
      ++m_c;
    }
    return(*this);
  }

//*********

  const_column_iterator &operator--() {
    if((m_c==0)&&(m_r==0))
      return(*this);
    if(m_r==0) {
      m_r=m_matrix->m_rows-1;
      --m_c;
    }
    else
      --m_r;
    return(*this);
  }

//*********

  const_column_iterator operator++(int) {
    const_column_iterator tmp=*this;
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

```
    ++(*this);
    return(tmp);
  }

//*********

  const_column_iterator operator--(int) {
    const_column_iterator tmp=*this;
    --(*this);
    return(tmp);
  }

//*********

  const Type &operator*() {
    return((*m_matrix)[m_r][m_c]);
  }

//*********

  const Type *operator->() {
    return(&(*m_matrix)[m_r][m_c]);
  }

//*********

  bool operator==(const_column_iterator &i) const {
    return((m_matrix==i.m_matrix)&&(m_c==i.m_c)&&(m_r==i.m_r));
  }

//*********

  bool operator!=(const_column_iterator &i) const {
    return((m_matrix!=i.m_matrix)||(m_c!=i.m_c)||(m_r!=i.m_r));
  }
};
```

Listing 123: Die const-Iteratoren von CMatrix (Forts.)

Iterator-Methoden

Damit zu einer Matrix auch Iteratoren erzeugt werden können, benötigen wir die obligatorischen `begin`- und `end`-Methoden.

Wir wollen aber auch hier einen Schritt weiter gehen als von der STL vorgelebt und zusätzliche Methoden implementieren, die Iteratoren für bestimmte Spalten erzeugen.

Ein Beispiel finden Sie im Anschluss an die Methoden.

```
/*
** begin & end für row_iterator
*/

row_iterator row_begin() {
  return(row_iterator(*this,0,0));
}

//*********

row_iterator row_end() {
  return(row_iterator(*this,m_rows,0));
}

//*********

row_iterator row_begin(size_type r) {
  return(row_iterator(*this,r,0));
}

//*********

row_iterator row_end(size_type r) {
  return(row_iterator(*this,r+1,0));
}

/*
** begin & end für const_row_iterator
*/

const_row_iterator row_begin() const {
  return(const_row_iterator(*this,0,0));
}

//*********
```

Listing 124: Die Iterator-Methoden von CMatrix

```
const_row_iterator row_end() const {
  return(const_row_iterator(*this,m_rows,0));
}

//*********

const_row_iterator row_begin(size_type r) const {
  return(const_row_iterator(*this,r,0));
}

//*********

const_row_iterator row_end(size_type r) const {
  return(const_row_iterator(*this,r+1,0));
}

/*
** begin & end für column_iterator
*/

column_iterator column_begin() {
  return(column_iterator(*this,0,0));
}

//*********

column_iterator column_end() {
  return(column_iterator(*this,0,m_columns));
}

//*********

column_iterator column_begin(size_type c) {
  return(column_iterator(*this,0,c));
}

//*********

column_iterator column_end(size_type c) {
  return(column_iterator(*this,0,c+1));
```

Listing 124: Die Iterator-Methoden von CMatrix (Forts.)

```
}

/*
** begin & end für const_column_iterator
*/

const_column_iterator column_begin() const {
  return(const_column_iterator(*this,0,0));
}

//*********

const_column_iterator column_end() const {
  return(const_column_iterator(*this,0,m_columns));
}

//*********

const_column_iterator column_begin(size_type c) const {
  return(const_column_iterator(*this,0,c));
}

//*********

const_column_iterator column_end(size_type c) const {
  return(const_column_iterator(*this,0,c+1));
}
```

Listing 124: Die Iterator-Methoden von CMatrix (Forts.)

Sie finden die Klasse `CMatrix` auf der CD in der `CMatrix.h`-Datei.

Wollen wir beispielsweise aus einer 5*5-Matrix `oMatrix` die zweite Spalte in einen Vektor `oVector` kopieren, dann sieht das wie folgt aus:

```
CMatrix<int> oMatrix(5,5);
vector<int> oVector;
copy(oMatrix.column_begin(1),
     oMatrix.column_end(1),
     back_inserter(oVector));
```

Wollten wir die dritte und vierte Zeile in den Vektor kopieren, geschähe dies so:

```
copy(oMatrix.row_begin(2),
     oMatrix.row_end(3),
     back_inserter(oVector));
```


38 Wie kann ein binärer Suchbaum implementiert werden?

Im Gegensatz zu einer linearen Liste, bei der jeder Knoten nur einen Nachfolger haben kann, zeichnet sich ein Suchbaum dadurch aus, dass ein Knoten maximal zwei Nachfolger besitzen darf. Abbildung 26 zeigt ein Beispiel eines Baums, der darüber hinaus noch vollständig ist.

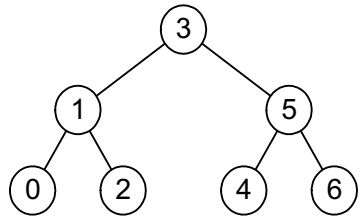


Abbildung 26: Ein Suchbaum

Die Besonderheit solcher Bäume liegt in der Sortierung eines Baums. In unserem Fall gilt die Regel, dass der linke Teilbaum eines Knotens nur Knoten enthält, deren Wert kleiner ist als der Vater des Teilbaums.

Einige STL-Container basieren auf Bäumen (z.B. `set`, `multiset`, `map`, `multimap`), allerdings handelt es sich dabei um ausgeglichene Bäume, denn bei gleicher Knotenanzahl ist der Baum mit der geringsten Höhe für Such-Operationen am effizientesten.

In Abbildung 27 sehen wir einen Baum, dessen Gesamthöhe durch eine Umstrukturierung (in diesem Fall eine Links-Rotation) um eins vermindert wird.

Allerdings geht dabei die ursprüngliche Baumstruktur verloren. Deswegen erzeugt die folgende Implementierung einen Baum, dessen Struktur nicht aus Effizienz-Gründen verändert wird.

Die Implementierung eines höhenbalancierten Baumes finden Sie in Rezept 43.

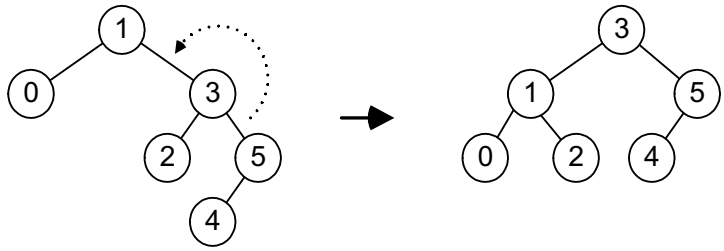


Abbildung 27: Höhenausgleichende Maßnahmen bei Bäumen

CKnot

Um einen einzelnen Knoten des Baumes repräsentieren zu können, legen wir eine Klasse `CKnot` an. Ein `CKnot`-Objekt besitzt Zeiger auf die beiden Söhne und den Vater. Zusätzlich beinhaltet das Objekt mit `m_data` das zu verwaltende Datenobjekt.

Abbildung 28 stellt die Zeigerstruktur grafisch dar.

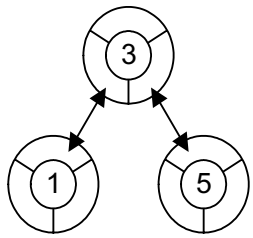


Abbildung 28: Die Zeigerstruktur der CKnot-Objekte

```
class CKnot {
public:
  CKnot *m_father;
  CKnot *m_left;
  CKnot *m_right;
  value_type m_data;

//*********

  CKnot(CKnot *fa, CKnot *li, CKnot *re)
  :m_father(fa),m_left(li),m_right(re)
  { }
```

Listing 125: Die Klasse CKnot

```
//*********

  CKnot(CKnot *fa, CKnot *li, CKnot *re, const value_type &v)
  :m_father(fa),m_left(li),m_right(re),m_data(v)
  {}

//*********

  virtual ~CKnot()
  {}
};
```

Listing 125: Die Klasse CKnot (Forts.)

CTree

Die Klasse `CTree` übernimmt die Verwaltung der Baumstruktur.

Sie wird als Template definiert, wobei ein `CTree`-Objekt an ein Objekt gebunden wird, das die Eigenschaften der zu verwaltenden Daten spezifiziert. Diese `Traits` besprechen wir später bei den konkreten Klassen `CSetTree` und `CMapTree`.

Klassendefinition

Abgesehen von den einzelnen Typendeklarationen, die zum größten Teil aus dem `Traits`-Objekt übernommen werden, besitzt die Klasse ein Attribut für die Wurzel des Baums und eins für die Anzahl der Objekte im Baum.

Die trivialen Methoden `size` und `empty` sind bei der Klassendefinition gleich mit aufgeführt.

```
template<typename Traits>
class CTree {
public:
  typedef CTree<Traits> my_type;
  typedef typename Traits::value_type value_type;
  typedef typename Traits::key_type key_type;
  typedef unsigned long size_type;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
```

Listing 126: Die Klassendefinition von CTree

```
protected:
  class CKnot { /* … */ };

  CKnot *m_root;
  size_type m_size;

//*********

  size_type size() const  {
    return(m_size);
  }

//*********

  bool empty() const {
    return(m_size==0);
  }
};
```

Listing 126: Die Klassendefinition von CTree (Forts.)

Konstruktoren

Der einfachste Konstruktor – der Standard-Konstruktor – erzeugt einen leeren Baum:

```
CTree()
: m_root(0), m_size(0)
{}
```

Listing 127: Der Standard-Konstruktor von CTree

Der Kopier-Konstruktor kopiert einen Baum durch Einsatz der `operator=`-Methode:

```
CTree(const CTree &t)
: m_root(0), m_size(0) {
  *this=t;
}
```

Listing 128: Der Kopier-Konstruktor von CTree

Der aufwändigste Konstruktor erzeugt den Baum aus Objekten eines Bereichs, der mit Iteratoren definiert wird. Für das Einfügen der Objekte in den Baum wird die `insert`-Methode benutzt:

```
template<typename Input>
CTree(Input beg, Input end)
: m_root(0), m_size(0) {
  insert(beg, end);
}
```

Listing 129: Ein weiterer Konstruktor für CTree

Destruktor

Der Destruktor löscht alle Knoten des Baums durch Aufruf der `deleteKnot`-Methode für die Wurzel des Baums:

```
virtual ~CTree() {
  if(m_root)
    deleteKnot(m_root);
}
```

Listing 130: Der Destruktor von CTree

deleteKnot

Die geschützte Methode `deleteKnot` löscht einen Knoten mitsamt der eventuell daran hängenden Teilbäume:

```
void deleteKnot(CKnot *kn) {
  if(kn) {
    if(kn->m_left) deleteKnot(kn->m_left);
    if(kn->m_right) deleteKnot(kn->m_right);
    delete(kn);
  }
}
```

Listing 131: Die geschützte Methode deleteKnot von CTree

Zuweisungsoperator

Der Zuweisungsoperator kopiert einen Baum, indem er zuerst die Wurzel kopiert und die weitere Arbeit dann der Methode `copySons` überlässt:

```
CTree &operator=(const CTree &t) {
  if(&t!=this) {

/*
** Alle Knoten des Baums löschen
*/
    deleteKnot(m_root);
    m_size=t.m_size;

/*
** Besitzt Quellbaum Knoten?
** = Wurzelknoten kopieren und alle weiteren Knoten
** durch Aufruf von copySons kopieren
*/
    if(t.m_root) {
      m_root=new CKnot(0,0,0,t.m_root->m_data);
      copySons(m_root, t.m_root);
    }
  }
  return(*this);
}
```

Listing 132: Der Zuweisungsoperator von CTree

copySons

Die private Methode `copySons` kopiert die Söhne eines Knotens und ruft für sie `copySons` erneut auf:

```
void copySons(CKnot* d, CKnot *s) {
  if(d&&s) {

/*
** Linken Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
```

Listing 133: Die private Methode copySons von CTree

```
    if(s->m_left) {
      d->m_left=new CKnot(d,0,0,s->m_left->m_data);
      copySons(d->m_left,s->m_left);
    }

/*
** Rechten Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_right) {
      d->m_right=new CKnot(d,0,0,s->m_right->m_data);
      copySons(d->m_right,s->m_right);
    }
  }
}
```

Listing 133: Die private Methode copySons von CTree (Forts.)

insert

Mit den verschiedenen `insert`-Methoden können Objekte in den Baum eingefügt werden.

Die einfachste Variante fügt ein Objekt in einen Baum ein und liefert die Position im Baum als Iterator-Position zurück. Sie benutzt dazu die `insert`-Methode zum Einfügen von Knoten.

Die Iteratoren werden in den nächsten Rezepten besprochen.

```
virtual inorder_iterator insert(const value_type &v) {

/*
** Datenobjekt in einen Knoten packen
*/
   CKnot *kn=new CKnot(0,0,0,v);

/*
** Knoten in Baum einfügen
*/
  insert(kn);
```

Listing 134: Eine insert-Methode von CTree

```
/*
** Position als Iterator zurückgeben
*/
  return(_iterator(kn,this,0));
}
```

Listing 134: Eine insert-Methode von CTree (Forts.)

Um eine `insert`-Methode zu besitzen, die »kompatibel« mit den `insert`-Methoden der STL-Baum-Container ist, fügen wir die folgende Variante hinzu.

Es ist selbstverständlich, dass die übergebene Iterator-Position ignoriert werden muss, weil das Datenobjekt durch die engen Sortierkriterien des Baums nicht frei im Baum platziert werden kann.

```
inorder_iterator insert(_iterator i, const value_type &v) {
  return(insert(v));
}
```

Listing 135: Eine insert-Methode von CTree

Die folgende `insert`-Methode fügt Objekte in einem durch Iteratoren definierten Bereich in den Baum ein:

```
template<typename Input>
void insert(Input beg, Input end) {
  while(beg!=end)
    insert(*(beg++));
}
```

Listing 136: Eine insert-Methode von CTree

Zum Schluss wird die `insert`-Methode vorgestellt, die den Hauptteil der Einfügearbeit leisten muss, denn sie fügt den Knoten in die Baum-Struktur ein:

```
void insert(CKnot *kn) {

/*
```

Listing 137: Die insert-Methode für Knoten

```
** Baum leer?
** => Einzufügender Knoten wird die Wurzel
*/
  if(!m_root) {
    m_root=kn;
    m_size++;
    return;
  }

  CKnot *cur=m_root;
  while(cur) {
    if(Traits::gt(Traits::getKey(cur->m_data),
                  Traits::getKey(kn->m_data))) {

/*
** Einzufügender Knoten kleiner als aktueller Knoten?
** => Falls vorhanden, im linken Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten linker
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_left) {
        cur->m_left=kn;
        kn->m_father=cur;
        m_size++;
        return;
      }
      else {
        cur=cur->m_left;
      }
    }

/*
** Einzufügender Knoten größer/gleich dem aktuellen Knoten?
** => Falls vorhanden, im rechten Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten rechter
**    Sohn des aktuellen Knotens
*/
    else {
      if(!cur->m_right) {
        cur->m_right=kn;
        kn->m_father=cur;
```

Listing 137: Die insert-Methode für Knoten (Forts.)

```
        m_size++;
        return;
      }
      else {
        cur=cur->m_right;
      }
    }
  }
}
```

Listing 137: Die insert-Methode für Knoten (Forts.)

erase

Mit den `erase`-Methoden können Knoten aus dem Baum entfernt werden.

Die erste Variante bekommt einen Schlüssel übergeben und löscht alle Knoten mit diesem Schlüssel. Sie liefert die Anzahl der gelöschten Knoten zurück:

```
virtual size_type erase(const key_type &k) {

/*
** Ersten Knoten mit Schlüssel k finden
*/
  CKnot *kn=findKnot(k);

/*
** Keine Knoten vorhanden?
** => Kein Knoten gelöscht
*/
  if(!kn)
    return(0);

/*
** Inorder-Iterator auf Knoten hinter dem zu löschenden
** Knoten setzen
*/
  inorder_iterator i=inorder_iterator(kn,this,0);
  ++i;
  size_type a=0;
```

Listing 138: Eine erase-Methode für Schlüssel

```
/*
** So lange laufen, wie noch Knoten mit
** Schlüssel k existieren
*/
  do {

/*
** Schlüssel löschen, neuen Knoten aus Iterator
** holen und Iterator inkrementieren
*/
    erase(kn);
    a++;
    kn=i.m_knot;
    ++i;
  } while((kn)&&(Traits::eq(Traits::getKey(kn->m_data),k)));
  return(a);
}
```

Listing 138: Eine erase-Methode für Schlüssel (Forts.)

Die oben stehende Methode benutzt einen Inorder-Iterator, der nicht vollständig initialisiert ist, denn der dritte Konstruktor-Parameter definiert nicht wie vorgeschrieben den ersten Knoten des Baumes. Letzlich spielt es in diesem Fall keine Rolle, weil der Iterator nur inkrementiert wird und von außen nicht zugänglich ist. Es vermindert aber die Laufzeit, weil eben der erste Knoten nicht bestimmt werden muss.

Die folgende Methode bekommt einen Iterator als Position übergeben und löscht den entsprechenden Knoten. Die Funktion gibt den in Inorder-Reihenfolge dahinter liegenden Knoten als Iterator-Position zurück:

```
virtual inorder_iterator erase(_iterator i) {

/*
** Iterator-Position hinter zu löschendem
** Knoten ermitteln
** (Ohne Anfangs-Position für Iterator zu bestimmen)
*/
  inorder_iterator io(i.m_knot,this,0);
  ++io;
```

Listing 139: Eine erase-Methode für Iteratoren

```
/*
** Konnte Knoten gelöscht werden?
** => Position hinter gelöschtem Knoten zurückgeben
** Andernfalls End-Position zurückgeben
*/
  if(erase(i.m_knot))
    return(_iterator(io.m_knot,this,0));
  else
    return(inorder_end());
}
```

Listing 139: Eine erase-Methode für Iteratoren (Forts.)

Die nächste `erase`-Methode löscht einen durch Iteratoren definierten Bereich. Der zweite Iterator muss dazu bezogen auf die Inorder-Reihenfolge hinter dem ersten Iterator liegen.

```
virtual inorder_iterator erase(inorder_iterator beg,
                               inorder_iterator end) {
  while((beg!=inorder_end())&&(beg!=end)) {
    CKnot *kn=beg.m_knot;
    ++beg;
    erase(kn);
  }
  return(beg);
}
```

Listing 140: Eine erase-Methode für Bereiche

Im Folgenden ist die `erase`-Methode aufgeführt, die einen Knoten tatsächlich aus dem Baum entfernt.

Dabei ist gewährleistet, dass alle Iterator-Positionen, die in Inorder-Reihenfolge auf Knoten hinter dem zu löschenden Knoten verweisen, gültig bleiben.

Aus dieser Garantie schöpfen jedoch nur die internen Methoden einen Vorteil. Für externe Einsätze gilt die alte Iterator-Regel, dass die Gültigkeit von Iteratoren nach Einfüge- oder Löschoperationen nicht gewährleistet wird.

```
bool erase(CKnot *cur) {
  if(!cur) return(false);

  CKnot *father=cur->m_father;

/*
** Zu löschender Knoten hat keine Söhne?
** => Kann problemlos gelöscht werden
*/
  if((!cur->m_left)&&(!cur->m_right)) {

/*
** Zu löschender Knoten ist die Wurzel?
** => Baum leer
*/
    if(cur==m_root) {
      m_root=0;
      delete(cur);
      m_size--;
      return(true);
    }

/*
** Zu löschender Knoten ist nicht die Wurzel?
** => Vater muss berücksichtigt werden
*/
    else {

/*
** Je nachdem, ob zu löschender Knoten der linke oder
** rechte Sohn des Vaters ist, muss entsprechender Sohn
** des Vaters auf 0 gesetzt werden.
*/
      if(father->m_left==cur) {
        father->m_left=0;
      }
      else {
        father->m_right=0;
      }
      delete(cur);
      m_size--;
      return(true);
    }
```

```
  }

/*
** Besitzt zu löschender Knoten zwei Söhne?
** => Zu löschenden Knoten durch symmetrischen
**    Vorgänger ersetzen und symmetrischen Vorgänger
**    löschen
*/
  if((cur->m_left)&&(cur->m_right)) {
    CKnot *sys=symmetricPred(cur);
    cur->m_data=sys->m_data;
    return(erase(sys));
  }

/*
** Besitzt zu löschender Knoten nur einen Sohn?
** => Sohn des zu löschenden Knotens wird Sohn vom
** Vater des zu löschenden Knotens
*/
  CKnot *son;
  if(cur->m_left)
    son=cur->m_left;
  else
    son=cur->m_right;

  if(cur!=m_root) {
    son->m_father=father;
    if(father->m_left==cur) {
      father->m_left=son;
    }
    else {
      father->m_right=son;
    }
  }

/*
** Ist zu löschender Sohn die Wurzel?
** => Sohn des zu löschenden Knotens wird Wurzel
*/
  else {
    son->m_father=0;
    m_root=son;
  }
```

```
  delete(cur);
  m_size--;
  return(true);
}
```

findFirstKnot

Die geschützte Methode `findFirstKnot` sucht bezogen auf die Inorder-Reihenfolge den ersten Knoten des Baumes mit übergebenem Schlüssel.

```
CKnot *findFirstKnot(const key_type &k) const {

/*
** Baum leer?
** => Knoten nicht gefunden
*/
  if(!m_root) return(0);

/*
** Obersten Knoten mit Schlüssel gleich k suchen
*/
  CKnot *cur=m_root;
  while(cur&&(Traits::ne(Traits::getKey(cur->m_data),k))) {
    if(Traits::lt(k,Traits::getKey(cur->m_data)))
      cur=cur->m_left;
    else
      cur=cur->m_right;
  }

/*
** Um ersten Knoten mit Schlüssel gleich k zu finden
** So lange im linken Ast hinabsteigen, wie Schlüssel gleich k
*/
  while((cur)&&(cur->m_left)&&
        (Traits::eq(Traits::getKey(cur->m_left->m_data),k)))
    cur=cur->m_left;

/*
** Überprüfen, ob symmetrische Vorgänge mit Schlüssel
** gleich k existieren
```

Listing 141: Die geschützte Methode findFirstKnot von CTree

```
*/
  if(cur) {
    CKnot *pred=symmetricPred(cur);
    while((pred)&&(Traits::eq(Traits::getKey(pred->m_data),k))) {
      cur=pred;
      pred=symmetricPred(cur);
    }
  }
  return(cur);
}
```

Listing 141: Die geschützte Methode findFirstKnot von CTree (Forts.)

findLastKnot

Diese Methode findet, bezogen auf die Inorder-Reihenfolge, den letzten Knoten, der gleich dem übergebenen Schlüssel ist:

```
CKnot *findLastKnot(const key_type &k) const{

/*
** Baum leer?
** => Knoten nicht gefunden
*/
  if(!m_root) return(0);

/*
** Obersten Knoten mit Schlüssel gleich k suchen
*/
  CKnot *cur=m_root;
  while(cur&&(Traits::ne(Traits::getKey(cur->m_data),k))) {
    if(Traits::lt(k,Traits::getKey(cur->m_data)))
      cur=cur->m_left;
    else
      cur=cur->m_right;
  }

/*
** Um letzten Knoten mit Schlüssel gleich k zu finden
** So lange im rechten Ast hinabsteigen wie Schlüssel gleich k
*/
```

Listing 142: Die geschützte Methode getLastKnot von CTree

```
  while((cur)&&(cur->m_right)&&
        (Traits::eq(Traits::getKey(cur->m_right->m_data),k)))
    cur=cur->m_right;

/*
** Überprüfen, ob symmetrische Nachfolger mit Schlüssel
** gleich k existieren
*/
  if(cur) {
    CKnot *succ=symmetricSucc(cur);
    while((succ)&&(Traits::eq(Traits::getKey(succ->m_data),k))) {
      cur=succ;
      succ=symmetricSucc(cur);
    }
  }
  return(cur);
}
```

Listing 142: Die geschützte Methode getLastKnot von CTree (Forts.)

symmetricPred

Diese Methode findet den symmetrischen Vorgänger eines Knotens.

```
CKnot *symmetricPred(CKnot *kn) const{
  CKnot *cur=kn->m_left;
  if(!cur)
    return(0);
  while(cur->m_right)
    cur=cur->m_right;
  return(cur);

}
```

Listing 143: Die geschützte Methode symmetricPred

symmetricSucc

Das Gegenstück zu `symmetricPred` bildet die Methode `symmetricSucc`, die den symmetrischen Nachfolger eines Knotens ermittelt:

```
CKnot *symmetricSucc(CKnot *kn) const {
  CKnot *cur=kn->m_right;
  if(!cur)
    return(0);
  while(cur->m_left)
    cur=cur->m_left;
  return(cur);
}
```

Listing 144: Die geschützte Methode symmetricSucc

lower_bound

`lower_bound` ermittelt die Iterator-Position des ersten Elements aus einer Gruppe gleicher Elemente:

```
inorder_iterator lower_bound(const key_type &k) {
  return(_iterator(findFirstKnot(k),this,0));
}

//*********

const_inorder_iterator lower_bound(const key_type &k) const{
  return(_const_iterator(findFirstKnot(k),this,0));
}
```

Listing 145: Die lower_bound-Methoden von CTree

upper_bound

`upper_bound` ermittelt die Iterator-Position hinter dem letzten Element aus einer Gruppe gleicher Elemente:

```
inorder_iterator upper_bound(const key_type &k) {
  inorder_iterator i=_iterator(findLastKnot(k),this,0);
  ++i;
  return(i);
}

//*********
```

Listing 146: Die upper_bound-Methoden von Ctree

```
const_inorder_iterator upper_bound(const key_type &k) const {
  const_inorder_iterator i=_const_iterator(findLastKnot(k),this,0);
  ++i;
  return(i);
}
```

Listing 146: Die upper_bound-Methoden von Ctree (Forts.)

find

Die Methode findet das erste Element mit entsprechendem Schlüssel. Von der Funktionalität her identisch mit `lower_bound`.

```
inorder_iterator find(const key_type &k) {
  return(_iterator(findFirstKnot(k),this,0));
}

//*********

const_inorder_iterator find(const key_type &k) const {
  return(_const_iterator(findFirstKnot(k),this,0));
}
```

Listing 147: Die find-Methoden von CTree

front & back

```
value_type &front() {
  return(inorder_begin().m_knot->m_data);
}

//*********

const value_type &front() const {
  return(inorder_begin().m_knot->m_data);
}

//*********
```

Listing 148: Die Methoden front und back von CTree

```
value_type &back() {
  return(inorder_rbegin().m_knot->m_data);
}

//*********

const value_type &back() const{
  return(inorder_rbegin().m_knot->m_data);
}
```

Listing 148: Die Methoden front und back von CTree (Forts.)

pop_front & pop_back

Die Methoden `pop_front` und `pop_back` entfernen das erste, beziehungsweise das letzte Element des Baums (bezogen auf die Inorder-Reihenfolge):

```
virtual void pop_front() {
  if(!empty())
    erase(inorder_begin().m_knot);
}

//*********

virtual void pop_back() {
  if(!empty())
    erase(inorder_rbegin().m_knot);
}
```

Listing 149: Die Methoden pop_front und pop_back von CTree

push_front & push_back

Diese beiden Methoden sind nur aus Kompatibilitätsgründen vorhanden. Die tatsächliche Einfügeposition des Elements ist durch die Sortierung des Baums vorgegeben.

```
virtual void push_front(const value_type &v) {
  insert(v);
}
```

Listing 150: Die Methoden push_front und push_back von Ctree

```
//*********

virtual void push_back(const value_type &v) {
  insert(v);
}
```

Listing 150: Die Methoden push_front und push_back von Ctree (Forts.)

Sie finden die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

set_tree_traits

Der Baum ist durch die Verwendung der selbst zu definierenden Tree-Traits ausgesprochen flexible. Wir können mit ihm sowohl die Funktionalität eines `set` als auch einer `map` erzeugen.

Schauen wir uns zunächst die `set_tree_traits` an, die das Verhalten von Schlüsseldaten definieren, die nur aus einer Komponente bestehen.

`value_type` und `key_type` sind in diesem Fall identisch.

Die Methode `getKey` liefert das Objekt selbst zurück.

Die Vergleichs-Methoden lassen sich alle auf den ==-Vergleich und das Prädikat `Cmp` zurückführen. Standardmäßig wird für `Cmp` das Prädikat `less` verwendet.

```
template<typename VType, typename Cmp= std::less<VType> >
class set_tree_traits {
public:
  typedef VType value_type;
  typedef VType key_type;

//*********

  static bool eq(const key_type &k1, const key_type &k2) {
    return(k1==k2);
  }

//*********

  static bool lt(const key_type &k1, const key_type &k2) {
```

Listing 151: Die Klasse set_tree_traits

```
    return(Cmp()(k1,k2));
  }

//*********

  static bool gt(const key_type &k1, const key_type &k2) {
    return(lt(k2,k1));
  }

//*********

  static bool ne(const key_type &k1, const key_type &k2) {
    return(!eq(k1,k2));
  }

//*********

  static bool ge(const key_type &k1, const key_type &k2) {
    return(!lt(k1,k2));
  }

//*********

  static bool le(const key_type &k1, const key_type &k2) {
    return(!lt(k2,k1));
  }

//*********

  static const key_type &getKey(const value_type &v) {
    return(v);
  }
};
```

Listing 151: Die Klasse set_tree_traits (Forts.)

CSetTree

Die Klasse `CSetTree` kombiniert nun `CTree` und `set_tree_traits` zu einer funktionstüchtigen Klasse.

Klassendefinition

Das Template besitzt als Parameter den zu verwaltenden Datentyp und das für die Vergleiche eingesetzte Prädikat (im Normalfall less).

Der einfacheren Handhabung wegen werden noch die Typen `base_type` für den Basisklassen-Typ und `my_type` für den eigenen Typ deklariert.

```
template<typename VType, typename Cmp= std::less<VType> >
class CSetTree : public CTree<set_tree_traits<VType, Cmp> > {
public:
  typedef CTree<set_tree_traits<VType, Cmp> > base_type;
  typedef CSetTree<VType, Cmp> my_type;
};
```

Listing 152: Die Klassendefinition von CSetTree

Konstruktoren

Mit den Konstruktoren wird lediglich die Funktionalität von `CTree` durchgeschliffen:

```
CSetTree()
: base_type()
{}

//*********

template<typename Input>
CSetTree(Input beg, Input end)
: base_type(beg,end)
{ }

//*********

CSetTree(const CSetTree &t)
: base_type(t)
{ }
```

Listing 153: Die Konstruktoren von CSetTree

Sie finden die Klassen `set_tree_traits` und `CSetTree` auf der CD in der `CSetTree.h`-Datei.

map_tree_traits

Um den Baum wie eine `map` einsetzen zu können, implementieren wir die `map_tree_traits`, die als zu verwaltenden Datentyp ein `pair` definiert, welches aus einem Schlüssel und den tatsächlichen Daten besteht.

`value_type` definiert damit den Datentyp der Nutzdaten und `key_type` den Datentyp der Schlüsseldaten.

Die Methode `getKey` liefert den Schlüssel, also das erste Element von `pair`, zurück.

Die Vergleichs-Methoden lassen sich alle auf den ==-Vergleich und das Prädikat `Cmp` zurückführen. Standardmäßig wird für `Cmp` das Prädikat `less` verwendet.

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
class map_tree_traits {
public:
  typedef typename std::pair<KType, VType> value_type;
  typedef KType key_type;

//*********

  static bool eq(const key_type &k1, const key_type &k2) {
    return(k1==k2);
  }

//*********

  static bool lt(const key_type &k1, const key_type &k2) {
    return(Cmp()(k1,k2));
  }

//*********

  static bool gt(const key_type &k1, const key_type &k2) {
    return(lt(k2,k1));
  }

//*********
```

Listing 154: Die Klasse map_tree_traits

```
  static bool ne(const key_type &k1, const key_type &k2) {
    return(!eq(k1,k2));
  }

//*********

  static bool ge(const key_type &k1, const key_type &k2) {
    return(!lt(k1,k2));
  }

//*********

  static bool le(const key_type &k1, const key_type &k2) {
    return(!lt(k2,k1));
  }

//*********

  static const key_type &getKey(const value_type &v) {
    return(v.first);
  }
};
```

Listing 154: Die Klasse map_tree_traits (Forts.)

CMapTree

Analog zu `CSetTree` erzeugt `CMapTree` mit `CTree` und `map_tree_traits` einen funktionsfähigen Baum.

Klassendefinition

Das Template besitzt als Parameter die Datentypen von Schlüssel- und Nutzdaten und das für die Vergleiche eingesetzte Prädikat (im Normalfall less).

Der einfacheren Handhabung wegen werden noch die Typen `base_type` für den Basisklassen-Typ und `my_type` für den eigenen Typ deklariert.

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
```

Listing 155: Die Klassendefinition von CMapTree

```
class CMapTree : public CTree<map_tree_traits<KType, VType, Cmp> > {
public:
  typedef CTree<map_tree_traits<KType, VType, Cmp> > base_type;
  typedef CMapTree<KType, VType, Cmp> my_type;
};
```

Listing 155: Die Klassendefinition von CMapTree (Forts.)

Konstruktoren

Auch hier bestehen die Konstruktoren nur aus Aufrufen der Basisklassen-Konstruktoren:

```
CMapTree()
: base_type()
{}

//*********

template<typename Input>
CMapTree(Input beg, Input end)
: base_type(beg,end)
{ }

//*********

CMapTree(const CMapTree &t)
: base_type(t)
{ }
```

Listing 156: Die Konstruktoren von CMapTree

Sie finden die Klassen `map_tree_traits` und `CMapTree` auf der CD in der *CMapTree.h*-Datei.

39 Wie kann ein Binärbaum mit Iteratoren in Inorder-Reihenfolge durchlaufen werden?

Bevor wir einen konkreten Iterator für den Baum implementieren, legen wir für die wichtigsten Iterator-Arten, die in den folgenden Rezepten noch vorgestellt werden, eine Basisklasse `_iterator` an.

_iterator

Die Klasse `_iterator` besitzt alle für einen Iterator notwendigen Attribute. Um die üblichen Durchlauf-Reihenfolgen implementieren zu können, reicht ein bidirektionaler Iterator. Wir leiten unsere Klasse dazu vom `iterator`-Template der Standardbibliothek ab.

Damit außerhalb des Baums von der Basisklasse kein Exemplar erzeugt werden kann, kommt sie in den geschützten Bereich.

Klassendefinition

Die Basisklasse unserer Iteratoren besitzt folgende Attribute:

- `m_knot` – ein Verweis auf den aktuellen Knoten und damit auf das aktuelle Element, auf das der Iterator zeigt
- `m_tree` – ein Zeiger auf den Baum, zu dem der Iterator gehört
- `m_begin` – die Bestimmung der Anfangsposition kostet je nach Iterator mehr als O(1)-Zeit, deswegen wird der Anfang nicht immer neu bestimmt, sondern im Iterator gespeichert.

```
class _iterator;
friend class _iterator;
class _iterator
: public std::iterator<std::bidirectional_iterator_tag, value_type> {
public:
  CKnot *m_knot;
  CKnot *m_begin;
  CTree *m_tree;
};
```

Listing 157: Die Klassendefinition von _iterator

Konstruktoren

Als Konstruktoren werden der Standard-Konstruktor und ein Konstruktor für alle Attribute implementiert.

```
_iterator()
  :m_knot(0), m_begin(0), m_tree(0)
```

Listing 158: Die Konstruktoren von _iterator

```
{}

//*********

_iterator(CKnot *kn, CTree *tr, CKnot *be)
  : m_knot(kn), m_tree(tr), m_begin(be)
{}
```

Listing 158: Die Konstruktoren von _iterator (Forts.)

operator* & operator->

Auf das mit dem Iterator verknüpfte Baum-Element muss mit Dereferenzierung und dem Zeiger-Operator zugegriffen werden können:

```
reference operator*() {
  return(m_knot->m_data);
}

//*********

pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 159: operator und operator-> von _iterator*

Vergleichsoperatoren

Für bidirektionale Iteratoren müssen die Vergleichsoperatoren == und != zur Verfügung gestellt werden:

```
bool operator==(_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(_iterator &i) const {
```

Listing 160: Die Vergleichsoperatoren von _iterator

```
  return(m_knot!=i.m_knot);
}
```

Listing 160: Die Vergleichsoperatoren von _iterator (Forts.)

isBegin

Um die Überprüfung, ob der Iterator augenblicklich die Anfangsposition seiner Durchlauf-Reihenfolge besitzt, nicht immer über einen Vergleich mit der unter Umständen zeitaufwändigeren `begin`-Methode umsetzen zu müssen, wurde die Methode `isBegin` implementiert.

Sie liefert `true` zurück, wenn der Iterator die Anfangsposition besitzt, andernfalls liefert sie `false`.

```
bool isBegin() const {
  return((m_knot!=0)&&(m_knot==m_begin));
}
```

Listing 161: Die Methode isBegin von _iterator

inorder_iterator

Mit dem `inorder_iterator` wird die Durchlaufreihenfolge Inorder implementiert. Sie entspricht der Sortierung des Baums und lässt sich rekursiv als linker Teilbaum – Knoten – rechter Teilbaum beschreiben.

Für den Baum in Abbildung 26 auf S. 155 beträgt die Inorder-Reihenfolge 0, 1, 2, 3, 4, 5, 6.

Klassendefinition

`inorder_iterator` wird von `_iterator` abgeleitet;

```
class inorder_iterator;
friend class inorder_iterator;
class inorder_iterator : public my_type::_iterator {
};
```

Listing 162: Die Klassendefinition von inorder_iterator

Konstruktoren

Zum einen werden die Iteratoren der Basisklasse durchgeschliffen.

Zusätzlich werden noch ein Kopier-Konstruktor und ein Konstruktor für Basisklassen-Objekte implementiert, die beide auf den Zuweisungsoperator der Klasse zurückgreifen.

```
inorder_iterator()
: _iterator(0,0,0)
{}

//*********

inorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

inorder_iterator(const inorder_iterator &i){
  *this=i;
}

//*********

inorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 163: Die Konstruktoren von inorder_iterator

Zuweisungsoperatoren

Der Zuweisungsoperator für Basisklassen-Objekte muss das Attribut `m_begin` neu bestimmen, weil dies unter Umständen gar nicht definiert wurde oder den Beginn einer anderen Durchlauf-Reihenfolge beinhaltet.

```
inorder_iterator &operator=(const inorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 164: Die Zuweisungsoperatoren von inorder_iterator

```
//*********

inorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->inorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 164: Die Zuweisungsoperatoren von inorder_iterator (Forts.)

operator++

Das erste Element der Durchlauf-Reihenfolge wird ermittelt, indem immer weiter im linken Ast hinabgestiegen wird, bis ein Knoten keinen linken Sohn mehr hat. Das übernimmt die Baum-Methode `inorder_begin`.

Von nun an werden zur Ermittlung des Nachfolgeknotens folgende Fälle unterschieden:

- Die Ende-Position ist erreicht: nichts machen.
- Der aktuelle Knoten besitzt einen rechten Sohn: symmetrischen Nachfolger bestimmen.
- Der aktuelle Knoten besitzt keinen rechten Sohn: so lange den Baum hochsteigen, bis der Vater des linken Astes erreicht ist.

```
inorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Existiert rechter Sohn?
** => Den linken Ast des rechten Sohns
**    komplett hinabsteigen
```

Listing 165: Die operator++-Methoden von inorder_iterator

```
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }

/*
** Existiert kein rechter Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der linke Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}

//*********

inorder_iterator operator++(int) {
  inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 165: Die operator++-Methoden von inorder_iterator (Forts.)

operator--

Die Funktionalität der Methode `operator--` ist identisch mit der `operator++`-Methode des Reverse-Iterators und wird dort erklärt.

```
inorder_iterator &operator--() {

/*
```

Listing 166: Die Methoden operator-- von inorder_iterator

```
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while(m_knot->m_right)
        m_knot=m_knot->m_right;
    }
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Existiert linker Sohn?
** => Den rechten Ast des linken Sohns
**    komplett hinabsteigen
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }

/*
** Existiert kein linker Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der rechte Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
```

Listing 166: Die Methoden operator-- von inorder_iterator (Forts.)

```
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

inorder_iterator operator--(int) {
  inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 166: Die Methoden operator-- von inorder_iterator (Forts.)

Iterator-Methoden

Die folgenden Methoden werden der Klasse `CTree` hinzugefügt, um Inorder-Iteratoren erzeugen zu können:

```
inorder_iterator inorder_begin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_left)
      kn=kn->m_left;
  }
  return(inorder_iterator(kn,this,kn));
}

//*********

inorder_iterator inorder_end() {
  return(inorder_iterator(0,this,0));
}
```

Listing 167: Die Iterator-Methoden für inorder_iterator

_const_iterator

Um auch Iteratoren auf konstante Bäume einsetzen zu können, implementieren wir eine geschützte Basisklasse `_const_iterator`, die der Klasse `_iterator` recht ähnlich ist.

Klassendefinition

Die Besonderheit des konstanten Iterators liegt im Zeiger auf einen konstanten Baum:

```
class _const_iterator;
friend class _const_iterator;
class _const_iterator
: public std::iterator<std::bidirectional_iterator_tag, value_type> {
public:
  CKnot *m_knot;
  CKnot *m_begin;
  const CTree *m_tree;
};
```

Listing 168: Die Klassendefinition von _const_iterator

Konstruktoren

Die Konstruktoren müssen berücksichtigen, dass der Baum konstant ist:

```
_const_iterator()
  :m_knot(0), m_begin(0), m_tree(0)
{}

//*********

_const_iterator(CKnot *kn, const CTree *tr, CKnot *be)
  : m_knot(kn), m_tree(tr), m_begin(be)
{}
```

Listing 169: Die Konstruktoren von _const_iterator

operator* & operator->

Der Zugriff über die Operatoren `*` und `->` muss einen Verweis auf ein konstantes Objekt zurückliefern.

```
const_reference operator*() {
  return(m_knot->m_data);
```

Listing 170: operator und operator-> von _const_iterator*

```
}

//*********

const_pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 170: operator und operator-> von _const_iterator (Forts.)*

Vergleichsoperatoren

```
bool operator==(_const_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(_const_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 171: Die Vergleichsoperatoren von _const_iterator

isBegin

```
bool isBegin() const {
  return((m_knot!=0)&&(m_knot==m_begin));
}
```

Listing 172: Die Methode isBegin von _const_iterator

const_inorder_iterator

Analog zu `inorder_iterator` wird `const_inorder_iterator` von `_const_iterator` abgeleitet.

Klassendefinition

```
class const_inorder_iterator;
friend class const_inorder_iterator;
class const_inorder_iterator : public my_type::_const_iterator {
};
```

Listing 173: Die Klassendefinition von const_inorder_iterator

Konstruktoren

```
const_inorder_iterator()
: _const_iterator(0,0,0)
{}

//*********

const_inorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
: _const_iterator(kn,tr,be)
{}

//*********

const_inorder_iterator(const const_inorder_iterator &i){
  *this=i;
}

//*********

const_inorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 174: Die Konstruktoren von const_inorder_iterator

Zuweisungsoperatoren

```
const_inorder_iterator &operator=(const const_inorder_iterator &i) {
  m_knot=i.m_knot;
```

Listing 175: Die Zuweisungsoperatoren von const_inorder_iterator

```
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_inorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->inorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 175: Die Zuweisungsoperatoren von const_inorder_iterator (Forts.)

operator++

```
const_inorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}

//*********

const_inorder_iterator operator++(int) {
```

Listing 176: Die Methoden operator++ von const_inorder_iterator

```
  const_inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 176: Die Methoden operator++ von const_inorder_iterator (Forts.)

operator--

```
const_inorder_iterator &operator--() {
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while(m_knot->m_right)
        m_knot=m_knot->m_right;
    }
    return(*this);
  }
  if(m_knot==m_begin)
    return(*this);

  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

const_inorder_iterator operator--(int) {
```

Listing 177: Die Methoden operator-- von const_inorder_iterator

```
  const_inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 177: Die Methoden operator-- von const_inorder_iterator (Forts.)

Iterator-Methoden

Die Iterator-Methoden werden wieder in `CTree` untergebracht.

```
const_inorder_iterator inorder_begin() const {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_left)
      kn=kn->m_left;
  }
  return(const_inorder_iterator(kn,this,kn));
}

const_inorder_iterator inorder_end() const {
  return(const_inorder_iterator(0,this,0));
}
```

Listing 178: Die Iterator-Methoden für const_inorder_iterator

reverse_inorder_iterator

Mit der Klasse `reverse_inorder_iterator` wird die umgekehrte Inorder-Durchlauf-Reihenfolge implementiert. Für Abbildung 26 wäre dies 6, 5, 4, 3, 2, 1, 0.

Klassendefinition

Die Klasse wird von `_iterator` abgeleitet:

```
class reverse_inorder_iterator;
friend class reverse_inorder_iterator;
class reverse_inorder_iterator : public my_type::_iterator {
};
```

Listing 179: Die Klassendefinition von reverse_inorder_iterator

Konstruktoren

Die Konstruktoren verfolgen den bewährten Ansatz, die Funktionalität der Zuweisungsoperatoren zu verwenden.

```
reverse_inorder_iterator()
: _iterator(0,0,0)
{}

//*********

reverse_inorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

reverse_inorder_iterator(const reverse_inorder_iterator &i){
  *this=i;
}

//*********

reverse_inorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 180: Die Konstruktoren von reverse_inorder_iterator

Zuweisungsoperatoren

Der Zuweisungsoperator für `_iterator`-Objekte muss nun `rbegin` verwenden.

```
reverse_inorder_iterator &operator=(const
                                    reverse_inorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 181: Die Zuweisungsoperatoren von reverse_inorder_iterator

```
//*********

reverse_inorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->inorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 181: Die Zuweisungsoperatoren von reverse_inorder_iterator (Forts.)

operator++

Das erste Element der Durchlauf-Reihenfolge wird ermittelt, indem immer weiter im linken Ast hinabgestiegen wird, bis ein Knoten keinen linken Sohn mehr hat. Das übernimmt die Baum-Methode `inorder_begin`.

Von nun an werden zur Ermittlung des Nachfolgeknotens folgende Fälle unterschieden:

- Die Ende-Position ist erreicht: nichts machen.
- Der aktuelle Knoten besitzt einen linken Sohn: symmetrischen Vorgänger bestimmen.
- Der aktuelle Knoten besitzt keinen linken Sohn: so lange den Baum hochsteigen, bis der Vater des rechten Astes erreicht ist.

```
reverse_inorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Existiert linker Sohn?
** => Den rechten Ast des linken Sohns
**    komplett hinabsteigen
*/
```

Listing 182: Die Methoden operator ++ von reverse_inorder_iterator

```
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
    while(m_knot->m_right)
      m_knot=m_knot->m_right;
  }

/*
** Existiert kein linker Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der rechte Sohn des Vaters ist.
**     Der Vater ist der neue Knoten.
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_left==son));
  }
  return(*this);
}

//*********

reverse_inorder_iterator operator++(int) {
  reverse_inorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 182: Die Methoden operator ++ von reverse_inorder_iterator (Forts.)

operator--

Die Methode `operator--` besitzt die gleiche Durchlauf-Reihenfolge wie `operator++` von `inorder_iterator`.

```
reverse_inorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator

```
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->inorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Existiert rechter Sohn?
** => Den linken Ast des rechten Sohns
**    komplett hinabsteigen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
    while(m_knot->m_left)
      m_knot=m_knot->m_left;
  }

/*
** Existiert kein rechter Sohn?
** ==> So lange hinaufsteigen, bis aktueller Knoten
**     der linke Sohn des Vaters ist.
**     Der Vater ist der neue Knoten
*/
  else {
    CKnot *son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&(m_knot->m_right==son));
  }
  return(*this);
}
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator (Forts.)

```
reverse_inorder_iterator operator--(int) {
  reverse_inorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 183: Die Methoden operator-- von reverse_inorder_iterator (Forts.)

Iterator-Methoden

Folgende Methoden werden der Klasse `CTree` mitgegeben, um Iteratoren des Typs `reverse_inorder_iterator` zu erzeugen:

```
reverse_inorder_iterator inorder_rbegin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_right)
      kn=kn->m_right;
  }
  return(reverse_inorder_iterator(kn,this,kn));
}

//*********

reverse_inorder_iterator inorder_rend() {
  return(reverse_inorder_iterator(0,this,0));
}
```

Listing 184: Die Iterator-Methoden für reverse_inorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

40 Wie kann ein Binärbaum mit Iteratoren in Preorder-Reihenfolge durchlaufen werden?

In Preorder-Reihenfolge wird von einem Baum zuerst die Wurzel, anschließend der linke Teilbaum und zum Schluss der rechte Teilbaum ausgegeben.

Damit ergibt der Baum in Abbildung 26 auf Seite 155 die Knoten-Reihenfolge 3, 1, 0, 2, 5, 4, 6.

preorder_iterator

Die Klasse `preorder_iterator` erlaubt es, einen Baum mit Iteratoren in der Preorder-Reihenfolge zu durchlaufen.

Klassendefinition

`preorder_iterator` wird von `_iterator` aus Rezept 39 abgeleitet, um Positionen zwischen den verschiedenen Iteratoren austauschen zu können.

```
class preorder_iterator;
friend class preorder_iterator;
class preorder_iterator : public my_type::_iterator {
};
```

Listing 185: Die Klassendefinition von preorder_iterator

Konstruktoren

Die Konstruktoren stellen die Funktionalität der Basisklasse zur Verfügung. Zusätzlich bieten zwei Konstruktoren unter Zuhilfenahme der Zuweisungsoperatoren noch die Möglichkeit, einen Iterator aus einem `_iterator`-Objekt oder einem `preorder_iterator`-Objekt (Kopier-Konstruktor) zu erzeugen.

```
preorder_iterator()
: _iterator(0,0,0)
{}

//*********

preorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

preorder_iterator(const preorder_iterator &i){
  *this=i;
}

//*********
```

Listing 186: Die Konstruktoren von preorder_iterator

```
preorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 186: Die Konstruktoren von preorder_iterator (Forts.)

Zuweisungsoperatoren

```
preorder_iterator &operator=(const preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

preorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->preorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 187: Die Zuweisungsoperatoren von preorder_iterator

operator++

Nachdem zum Beispiel mit der Methode `preorder_begin` ein gültiger Iterator erzeugt wurde, ermittelt die `operator++`-Methode den nächsten Knoten der Durchlauf-Reihenfolge durch Unterscheidung der folgenden Fälle:

- Linker Ast noch nicht durchlaufen: ersten Knoten des linken Astes bestimmen.
- Linker Ast durchlaufen, rechten Ast noch nicht: ersten Knoten des rechten Astes bestimmen.
- Beide Äste durchlaufen: im Baum so lange aufsteigen, wie sich der Iterator im rechten Teilbaum befindet oder er sich im linken Teilbaum befindet und kein rechter Ast vorhanden ist.

```
preorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Linker Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Linken Ast, aber rechten Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im linken Ast aufsteigt und
** kein rechter Ast verfügbar ist oder der Iterator im rechten
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
```

Listing 188: Die Methoden operator ++ von preorder_iterator

```
            (((son==m_knot->m_left)&&(!m_knot->m_right) )||
             (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

preorder_iterator operator++(int) {
  preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 188: Die Methoden operator ++ von preorder_iterator (Forts.)

operator--

`operator--` besitzt nahezu die gleiche Funktionsweise wie `operator++` des Reverse-Iterators:

```
preorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->preorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
    return(*this);
```

Listing 189: Die Methoden operator-- von preorder_iterator

```
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Linker Ast noch nicht abgearbeitet?
** => Linken Ast so weit wie möglich hinabsteigen,
**    wobei rechte Teiläste Priorität haben
*/
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
  }
  return(*this);
}

//*********

preorder_iterator operator--(int) {
  preorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 189: Die Methoden operator-- von preorder_iterator (Forts.)

Iterator-Methoden

Es folgen die beiden Methoden aus `CTree`, mit denen Preorder-Iteratoren erzeugt werden können:

```
preorder_iterator preorder_begin() {
  return(preorder_iterator(m_root,this, m_root));
}

//*********

preorder_iterator preorder_end() {
  return(preorder_iterator(0,this,0));
}
```

Listing 190: Die Iterator-Methoden für preorder_iterator

const_preorder_iterator

Der Vollständigkeit halber wird hier noch der `const_preorder_iterator` vorgestellt, der die Preorder-Traversion konstanter Bäume erlaubt:

Klassendefinition

`const_preorder_iterator` wird von `_const_iterator` aus Rezept 39 abgeleitet:

```
class const_preorder_iterator;
friend class const_preorder_iterator;
class const_preorder_iterator : public my_type::_const_iterator {
};
```

Listing 191: Die Klassendefinition von const_preorder_iterator

Konstruktoren

```
const_preorder_iterator()
  : _const_iterator(0,0,0)
{}

//*********
```

Listing 192: Die Konstruktoren von const_preorder_iterator

```
const_preorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
  : _const_iterator(kn,tr,be)
{}

//*********

const_preorder_iterator(const const_preorder_iterator &i){
  *this=i;
}

//*********

const_preorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 192: Die Konstruktoren von const_preorder_iterator (Forts.)

Zuweisungsoperatoren

```
const_preorder_iterator &operator=(const
                                   const_preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_preorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->const_preorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 193: Die Zuweisungsoperatoren von const_preorder_iterator

operator++

```
const_preorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }
  else {
    CKnot * son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_left)&&(!m_knot->m_right) )||
              (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

const_preorder_iterator operator++(int) {
  const_preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 194: Die Methoden operator ++ von const_preorder_iterator

operator--

```
const_preorder_iterator &operator--() {
  if(!m_knot) {
```

Listing 195: Die Methoden operator-- von const_preorder_iterator

```
    *this=m_tree->preorder_begin();
    m_knot=m_tree->m_root;
    if(m_knot) {
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
    return(*this);
  }

  if(m_knot==m_begin)
    return(*this);

  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
      }
    }
  }
  return(*this);
}

//*********

const_preorder_iterator operator--(int) {
  const_preorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 195: Die Methoden operator-- von const_preorder_iterator (Forts.)

Iterator-Methoden

Und zum guten Schluss die Iterator-erzeugenden Methoden aus `CTree`:

```
const_preorder_iterator preorder_begin() const {
  return(const_preorder_iterator(m_root,this, m_root));
}

//*********

const_preorder_iterator preorder_end() const {
  return(const_preorder_iterator(0,this,0));
}
```

Listing 196: Die Iterator-Methoden für const_preorder_iterator

reverse_preorder_iterator

Um einen Baum in umgekehrter Preorder-Reihenfolge durchlaufen zu können, was in Abbildung 26 auf Seite 155 der Reihenfolge 6, 4, 5, 2, 0, 1, 3 entspricht, fügen wir die Klasse `reverse_preorder_iterator` hinzu.

Klassendefinition

Wie alle nicht-konstanten Iteratoren bisher wird `reverse_preorder_iterator` von `_iterator` aus Rezept 39 abgeleitet:

```
class reverse_preorder_iterator;
friend class reverse_preorder_iterator;
class reverse_preorder_iterator : public my_type::_iterator {
};
```

Listing 197: Die Klassendefinition von reverse_preorder_iterator

Konstruktoren

```
reverse_preorder_iterator()
: _iterator(0,0,0)
{}
```

Listing 198: Die Konstruktoren von reverse_preorder_iterator

```
//*********

reverse_preorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: iterator(kn,tr,be)
{}

//*********

reverse_preorder_iterator(const reverse_preorder_iterator &i){
  *this=i;
}

//*********

reverse_preorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 198: Die Konstruktoren von reverse_preorder_iterator (Forts.)

Zuweisungsoperatoren

```
reverse_preorder_iterator &operator=(const reverse_preorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

reverse_preorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->preorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 199: Die Zuweisungsoperatoren von reverse_preorder_iterator

operator++

Um den nächsten Knoten zu bestimmen, ermittelt `operator++` zunächst den Vater des aktuellen Knotens.

Sollte der aktuelle Knoten der linke Sohn gewesen sein, dann ist der Vater der neue Knoten.

Sollte der aktuelle Knoten nicht der linke Sohn gewesen sein, so steigt `operator++` immer tiefer im linken Teilbaum hinab.

Sollte ein Knoten zwei Söhne haben, so wird immer der rechte genommen.

Der letzte auf diese Weise ermittelte Knoten ist der neue Knoten.

```
reverse_preorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Linker Ast noch nicht abgearbeitet?
** => Linken Ast so weit wie möglich hinabsteigen,
**    wobei rechte Teiläste Priorität haben
*/
    if((m_knot->m_left)&&((m_knot->m_left!=son))) {
      m_knot=m_knot->m_left;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_right)
          m_knot=m_knot->m_right;
        else
          m_knot=m_knot->m_left;
```

Listing 200: Die Methoden operator ++ von reverse_preorder_iterator

```
      }
    }
  }
  return(*this);
}

//*********

reverse_preorder_iterator operator++(int) {
  reverse_preorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 200: Die Methoden operator ++ von reverse_preorder_iterator (Forts.)

operator--

Diese Methode verfolgt das gleiche Prinzip wie `operator++` von `preorder_iterator`:

```
reverse_preorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->preorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Linker Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator

```
*/
  if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Linken Ast, aber rechten Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  else if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im linken Ast aufsteigt und
** kein rechter Ast verfügbar ist oder der Iterator im rechten
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_left)&&(!m_knot->m_right) )||
             (son==m_knot->m_right)));
    if(m_knot)
      m_knot=m_knot->m_right;
  }
  return(*this);
}

//*********

reverse_preorder_iterator operator--(int) {
  reverse_preorder_iterator tmp=*this;
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator (Forts.)

```
  --(*this);
  return(tmp);
}
```

Listing 201: Die Methoden operator-- von reverse_preorder_iterator (Forts.)

Iterator-Methoden

Mit den folgenden Methoden lassen sich über ein `CTree`-Objekt entsprechende Iteratoren erzeugen:

```
reverse_preorder_iterator preorder_rbegin() {
  CKnot *kn=m_root;
  if(kn) {
    while(kn->m_right)
      kn=kn->m_right;
  }
  return(reverse_preorder_iterator(kn,this, kn));
}

//*********

reverse_preorder_iterator preorder_rend() {
  return(reverse_preorder_iterator(0,this,0));
}
```

Listing 202: Die Iterator-Methoden für reverse_preorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

41 Wie kann ein Binärbaum mit Iteratoren in Postorder-Reihenfolge durchlaufen werden?

Die Postorder-Reihenfolge ist bezogen auf einen Knoten so definiert, dass zuerst der linke Teilbaum, dann der rechte Teilbaum und zum Schluss der Knoten selbst ausgegeben wird.

Der Baum in Abbildung 26 auf Seite 155 würde in Postorder 0, 2, 1, 4, 6, 5, 3 ausgegeben.

postorder_iterator

Für die Postorder-Funktionalität wird eine neue Klasse `postorder_iterator` angelegt.

Klassendefinition

Auch `postorder_iterator` wird von `_iterator` aus Rezept 39 abgeleitet.

```
class postorder_iterator;
friend class postorder_iterator;
class postorder_iterator : public my_type::_iterator {
};
```

Listing 203: Die Klassendefinition von postorder_iterator

Konstruktoren

Als Konstruktoren steht der Kopier-Konstruktor, ein Konstruktor für `_iterator`-Objekte und die beiden durchgeschliffenen Basisklassen-Konstruktoren zur Verfügung.

```
postorder_iterator()
: _iterator(0,0,0)
{}

//*********

postorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
: _iterator(kn,tr,be)
{}

//*********

postorder_iterator(const postorder_iterator &i){
  *this=i;
}

//*********

postorder_iterator(const _iterator &i){
```

Listing 204: Die Konstruktoren von postorder_iterator

```
  *this=i;
}
```

Listing 204: Die Konstruktoren von postorder_iterator (Forts.)

Zuweisungsoperatoren

Die Zuweisungsoperatoren, von denen auch zwei der Konstruktoren Gebrauch machen, weisen einem `postorder_iterator`-Objekt ein anderes `postorder_iterator`-Objekt oder ein `_iterator`-Objekt zu:

```
postorder_iterator &operator=(const postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

postorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->postorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 205: Die Zuweisungsoperatoren von postorder_iterator

operator++

Um den nächsten Knoten zu bestimmen, ermittelt `operator++` zunächst den Vater des aktuellen Knotens.

Sollte der aktuelle Knoten der rechte Sohn gewesen sein, dann ist der Vater der neue Knoten.

Sollte der aktuelle Knoten nicht der rechte Sohn gewesen sein, so steigt `operator++` immer tiefer im rechten Teilbaum hinab.

Sollte ein Knoten zwei Söhne haben, so wird immer der linke genommen.

Der letzte auf diese Weise ermittelte Knoten ist der neue Knoten.

```
postorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Rechter Ast noch nicht abgearbeitet?
** => Rechten Ast so weit wie möglich hinabsteigen,
**    wobei linke Teiläste Priorität haben
*/
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

postorder_iterator operator++(int) {
  postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 206: Die Methoden operator ++ von postorder_iterator

operator--

Die Methode `operator--` ist von der Grundidee her identisch mit `operator++` des Reverse-Iterators:

```
postorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    m_knot=m_tree->m_root;
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Rechter Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Rechten Ast, aber linken Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Beide Äste abgearbeitet?
```

Listing 207: Die Methoden operator-- von postorder_iterator

```
** => aufsteigen und nächsten Teilast finden
*/
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im rechten Ast aufsteigt und
** kein linker Ast verfügbar ist oder der Iterator im linken
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_right)&&(!m_knot->m_left) )||
             (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

postorder_iterator operator--(int) {
  postorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 207: Die Methoden operator-- von postorder_iterator (Forts.)

Iterator-Methoden

Um einen `postorder_iterator` zu erzeugen, werden die folgenden Methoden von `CTree` benutzt:

```
postorder_iterator postorder_begin() {
  CKnot *kn=m_root;
  if(kn) {
    while((kn->m_left)||(kn->m_right)) {
```

Listing 208: Die Iterator-Methoden für postorder_iterator

```
      if(kn->m_left)
        kn=kn->m_left;
      else
        kn=kn->m_right;
    }
  }
  return(postorder_iterator(kn,this,kn));
}

//*********

postorder_iterator postorder_end() {
  return(postorder_iterator(0,this,0));
}
```

Listing 208: Die Iterator-Methoden für postorder_iterator (Forts.)

const_postorder_iterator

Natürlich muss auch ein konstanter Baum in Postorder-Reihenfolge durchlaufen werden können, deswegen wird die Klasse `const_postorder_iterator` geschrieben.

Klassendefinition

Um austauschbar mit den anderen Iteratoren für konstante Bäume zu bleiben, wird `const_postorder_iterator` von `_const_iterator` aus Rezept 39 abgeleitet:

```
class const_postorder_iterator;
friend class const_postorder_iterator;
class const_postorder_iterator : public my_type::_const_iterator {
};
```

Listing 209: Die Klassendefinition von const_postorder_iterator

Konstruktoren

```
const_postorder_iterator()
: _const_iterator(0,0,0)
{}
```

Listing 210: Die Konstruktoren von const_postorder_iterator

```
//*********

const_postorder_iterator(CKnot *kn, const CTree *tr, CKnot *be)
: _const_iterator(kn,tr,be)
{}

//*********

const_postorder_iterator(const const_postorder_iterator &i){
  *this=i;
}

//*********

const_postorder_iterator(const _const_iterator &i){
  *this=i;
}
```

Listing 210: Die Konstruktoren von const_postorder_iterator (Forts.)

Zuweisungsoperatoren

```
const_postorder_iterator &operator=(const
                                    const_postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

const_postorder_iterator &operator=(const _const_iterator &i) {
  *this=i.m_tree->postorder_begin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 211: Die Zuweisungsoperatoren von const_postorder_iterator

operator++

```
const_postorder_iterator &operator++() {
  if(!m_knot)
    return(*this);

  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

const_postorder_iterator operator++(int) {
  const_postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 212: Die Methoden operator ++ von const_postorder_iterator

operator--

```
const_postorder_iterator &operator--() {
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    m_knot=m_tree->m_root;
    return(*this);
  }
```

Listing 213: Die Methoden operator-- von const_postorder_iterator

```
  if(m_knot==m_begin)
    return(*this);

  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }
  else {
    CKnot * son;
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
           (((son==m_knot->m_right)&&(!m_knot->m_left) )||
            (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

const_postorder_iterator operator--(int) {
  const_postorder_iterator tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 213: Die Methoden operator-- von const_postorder_iterator (Forts.)

Iterator-Methoden

```
const_postorder_iterator postorder_begin() const {
  CKnot *kn=m_root;
  if(kn) {
    while((kn->m_left)||(kn->m_right)) {
```

Listing 214: Die Iterator-Methoden für const_postorder_iterator

```
      if(kn->m_left)
        kn=kn->m_left;
      else
        kn=kn->m_right;
    }
  }
  return(const_postorder_iterator(kn,this,kn));
}

//*********

const_postorder_iterator postorder_end() const {
  return(const_postorder_iterator(0,this,0));
}
```

Listing 214: Die Iterator-Methoden für const_postorder_iterator (Forts.)

reverse_postorder_iterator

Wir wollen auch die Möglichkeit unterstützen, mit einem Reverse-Iterator den Baum in umgekehrter Postorder-Reihenfolge zu durchlaufen. Für unseren Baum in Abbildung 26 auf Seite 155 wäre dies 3, 5, 6, 4, 1, 2, 0.

Klassendefinition

Wir leiten `reverse_postorder_iterator` von der Klasse `_iterator` aus Rezept 39 ab:

```
class reverse_postorder_iterator;
friend class reverse_postorder_iterator;
class reverse_postorder_iterator : public my_type::_iterator {
};
```

Listing 215: Die Klassendefinition von reverse_postorder_iterator

Konstruktoren

```
reverse_postorder_iterator()
  : _iterator(0,0,0)
{}
```

Listing 216: Die Konstruktoren von reverse_postorder_iterator

```
//*********

reverse_postorder_iterator(CKnot *kn, CTree *tr, CKnot *be)
  : _iterator(kn,tr,be)
{}

//*********

reverse_postorder_iterator(const reverse_postorder_iterator &i){
  *this=i;
}

//*********

reverse_postorder_iterator(const _iterator &i){
  *this=i;
}
```

Listing 216: Die Konstruktoren von reverse_postorder_iterator (Forts.)

Zuweisungsoperatoren

```
reverse_postorder_iterator &operator=(const
                                reverse_postorder_iterator &i) {
  m_knot=i.m_knot;
  m_begin=i.m_begin;
  m_tree=i.m_tree;
  return(*this);
}

//*********

reverse_postorder_iterator &operator=(const _iterator &i) {
  *this=i.m_tree->postorder_rbegin();
  m_knot=i.m_knot;
  m_tree=i.m_tree;
  return(*this);
}
```

Listing 217: Die Zuweisungsoperatoren von reverse_postorder_iterator

operator++

Die Methode `operator++` ermittelt den nächsten Knoten der Durchlauf-Reihenfolge durch Unterscheidung der folgenden Fälle:

- Rechter Ast noch nicht durchlaufen: ersten Knoten des rechten Astes bestimmen.
- Rechten Ast durchlaufen, linken Ast noch nicht: ersten Knoten des linken Astes bestimmen.
- Beide Äste durchlaufen: im Baum so lange aufsteigen, wie sich der Iterator im linken Teilbaum befindet oder er sich im rechten Teilbaum befindet und kein linker Ast vorhanden ist.

```
reverse_postorder_iterator &operator++() {

/*
** Ende bereits erreicht?
** => Nichts machen
*/
  if(!m_knot)
    return(*this);

/*
** Rechter Ast noch nicht abgearbeitet?
** => Ersten Knoten des rechten Astes bestimmen
*/
  if(m_knot->m_right) {
    m_knot=m_knot->m_right;
  }

/*
** Rechten Ast, aber linken Ast noch nicht abgearbeitet?
** => Ersten Knoten des linken Astes bestimmen
*/
  else if(m_knot->m_left) {
    m_knot=m_knot->m_left;
  }

/*
** Beide Äste abgearbeitet?
** => aufsteigen und nächsten Teilast finden
*/
```

Listing 218: Die Methoden operator ++ von reverse_postorder_iterator

```
  else {
    CKnot * son;

/*
** Laufe so lange, wie Iterator im rechten Ast aufsteigt und
** kein linker Ast verfügbar ist oder der Iterator im linken
** Ast aufsteigt
*/
    do {
      son=m_knot;
      m_knot=m_knot->m_father;
    } while((m_knot)&&
            (((son==m_knot->m_right)&&(!m_knot->m_left) )||
             (son==m_knot->m_left)));
    if(m_knot)
      m_knot=m_knot->m_left;
  }
  return(*this);
}

//*********

reverse_postorder_iterator operator++(int) {
  reverse_postorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 218: Die Methoden operator ++ von reverse_postorder_iterator (Forts.)

operator--

Die Methode `operator--` verfolgt zur Ermittlung des nächsten Knotens eine ähnliche Strategie wie `operator++` von `postorder_iterator`:

```
reverse_postorder_iterator &operator--() {

/*
** Besitzt Iterator Endposition?
** => Letztes Element der Durchlauf-Reihenfolge bestimmen
*/
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator

```
  if(!m_knot) {
    *this=m_tree->postorder_begin();
    return(*this);
  }

/*
** Besitzt Iterator Anfangsposition?
** => Nichts machen
*/
  if(m_knot==m_begin)
    return(*this);

/*
** Vater ermitteln
*/
  CKnot *son=m_knot;
  m_knot=m_knot->m_father;
  if(m_knot) {

/*
** Rechter Ast noch nicht abgearbeitet?
** => Rechten Ast so weit wie möglich hinabsteigen,
**    wobei linke Teiläste Priorität haben
*/
    if((m_knot->m_right)&&((m_knot->m_right!=son))) {
      m_knot=m_knot->m_right;
      while((m_knot->m_left)||(m_knot->m_right)) {
        if(m_knot->m_left)
          m_knot=m_knot->m_left;
        else
          m_knot=m_knot->m_right;
      }
    }
  }
  return(*this);
}

//*********

reverse_postorder_iterator operator--(int) {
  reverse_postorder_iterator tmp=*this;
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator (Forts.)

```
  --(*this);
  return(tmp);
}
```

Listing 219: Die Methoden operator-- von reverse_postorder_iterator (Forts.)

Iterator-Methoden

Erzeugt wird ein `reverse_postorder_iterator`-Objekt mit folgenden Methoden, die in `CTree` zu finden sind:

```
reverse_postorder_iterator postorder_rbegin() {
  return(reverse_postorder_iterator(m_root,this,m_root));
}

//*********

reverse_postorder_iterator postorder_rend() {
  return(reverse_postorder_iterator(0,this,0));
}
```

Listing 220: Die Iterator-Methoden für reverse_postorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der *CTree.h*-Datei.

42 Wie kann ein Binärbaum mit Iteratoren in Levelorder-Reihenfolge durchlaufen werden?

Die Durchlauf-Reihenfolge Levelorder erfordert ein anderes Vorgehen als das der anderen Iteratoren. Bei ihr werden die Knoten Baumlevel für Baumlevel ausgegeben. Der Baum in Abbildung 26 auf S. 155 ergibt in Levelorder ausgegeben beispielsweise 3, 1, 5, 0, 2, 4, 6.

Während Durchlauf-Reihenfolgen wie Inorder, Preorder oder Postorder einer Tiefensuche ähneln, baut der Ansatz für eine Levelorder-Reihenfolge auf einer Breitensuche auf.

levelorder_iterator

Um die Levelorder-Funktionalität umsetzen zu können, schreiben wir eine Klasse `levelorder_iterator`.

Klassendefinition

Der Levelorder-Iterator benötigt mit Abstand den größten Speicherbereich. Um ihn dennoch so gering wie möglich zu halten, verzichten wir bei ihm auf operator-- und definieren ihn als Forward-Iterator.

Die Klasse besitzt folgende Attribute:

- `m_knot` – ein Verweis auf den aktuellen Knoten und damit auf das aktuelle Element, auf das der Iterator zeigt.
- `m_tree` – ein Zeiger auf den Baum, zu dem der Iterator gehört.
- `m_queue` – im konkreten Fall eine Deque, in der die Knoten des nächsten Levels gespeichert werden. (Notwendig für ein Breitensuche-Verhalten.)
- `m_level` – der Level, auf dem sich der aktuelle Knoten im Baum befindet.

```
class levelorder_iterator;
friend class levelorder_iterator;
class levelorder_iterator
: public std::iterator<std::forward_iterator_tag, value_type> {
private:
  CKnot *m_knot;
  CTree *m_tree;
  int m_level;
  std::deque<CKnot*> m_queue;
};
```

Listing 221: Die Klassendefinition von levelorder_iterator

Konstruktoren

Es steht sowohl ein Standard-Konstruktor zur Verfügung als auch ein von den Methoden `levelorder_begin` und `levelorder_end` verwendeter Konstruktor mit einem Knoten- und einem Baum-Objekt als Parameter.

```
levelorder_iterator()
:m_knot(0), m_tree(0),m_level(0) {
  m_queue.push_back(0);
}

//*********
```

Listing 222: Die Konstruktoren von levelorder_iterator

```
levelorder_iterator(CKnot *kn, CTree *tr)
: m_knot(kn), m_tree(tr),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 222: Die Konstruktoren von levelorder_iterator (Forts.)

Zuweisungsoperatoren

Als Zuweisungsoperator benutzen wir den standardmäßig vom Compiler zur Verfügung gestellten Zuweisungsoperator für flache Kopien.

operator* & operator->

```
reference operator*() {
  return(m_knot->m_data);
}

//*********

pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 223: Die Methoden operator und operator-> von levelorder_iterator*

Vergleichsoperatoren

```
bool operator==(levelorder_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(levelorder_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 224: Die Vergleichsoperatoren von levelorder_iterator

operator++

Bevor die `operator++`-Methode den nächsten Knoten aus der Queue holt, werden die Söhne des aktuellen Knotens an die Queue angehängt. Auf diese Weise liegen die Knoten Level für Level in der Queue.

Jedes Mal, wenn ein neuer Level »angebrochen« wird, fügen wir einen 0-Konten in die Queue ein, um später den Wechsel des Levels erkennen und `m_level` um eins erhöhen zu können.

Weil wir keinen Operator `--` implementieren, können die Knoten tatsächlich aus der Queue entfernt werden, sodass sich in der Queue maximal nur die Knoten von zwei Leveln befinden.

```
levelorder_iterator &operator++() {

/*
** Bevor nächster Knoten aus Queue geholt wird,
** die beiden Söhne (falls vorhanden) an die Queue
** anhängen
*/
  if(m_knot->m_left)
    m_queue.push_back(m_knot->m_left);
  if(m_knot->m_right)
    m_queue.push_back(m_knot->m_right);

/*
** Nächsten Knoten aus Queue holen
*/
  if(m_queue.size()!=0) {
    m_knot=m_queue.front();
    m_queue.pop_front();

/*
** Nächsten Level erreicht?
** => Neue Markierung in die Queue schieben und
**    Level um eins erhöhen
*/
    if(!m_knot) {
      m_level++;
      m_queue.push_back(0);
      m_knot=m_queue.front();
```

Listing 225: Die Methode operator++ von levelorder_iterator

```
      m_queue.pop_front();
    }
  }

/*
** Queue leer?
** => Ende des Baumes erreicht
*/
  else
    m_knot=0;
  return(*this);
}

//*********

levelorder_iterator operator++(int) {
  levelorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 225: Die Methode operator++ von levelorder_iterator (Forts.)

getLevel

Mit `getLevel` kann der Level des Knotens bestimmt werden, auf den der Iterator zeigt.

```
int getLevel() const {
  return(m_level);
}
```

Listing 226: Die getLevel-Methode von levelorder_iterator

Iterator-Methoden

Mit den Methoden `levelorder_begin` und `levelorder_end` kann über ein `CTree`-Objekt ein entsprechender Levelorder-Iterator erzeugt werden:

```
levelorder_iterator levelorder_begin() {
  return(levelorder_iterator(m_root,this));
}

//*********

levelorder_iterator levelorder_end() {
  return(levelorder_iterator(0,this));
}
```

Listing 227: Die Iterator-Methoden für levelorder_iterator

const_levelorder_iterator

`const_levelorder_iterator` implementiert einen Levelorder-Iterator für konstante Bäume.

Klassendefinition

Die Klasse besitzt nun einen Zeiger auf einen konstanten Baum:

```
class const_levelorder_iterator;
friend class const_levelorder_iterator;
class const_levelorder_iterator
: public std::iterator<std::forward_iterator_tag, value_type> {
private:
  CKnot *m_knot;
  const CTree *m_tree;
  int m_level;
  std::deque<CKnot*> m_queue;
};
```

Listing 228: Die Klassendefinition von const_levelorder_iterator

Konstruktoren

```
const_levelorder_iterator()
  :m_knot(0), m_tree(0),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 229: Die Konstruktoren von const_levelorder_iterator

```
//*********

const_levelorder_iterator(CKnot *kn, const CTree *tr)
  : m_knot(kn), m_tree(tr),m_level(0) {
  m_queue.push_back(0);
}
```

Listing 229: Die Konstruktoren von const_levelorder_iterator (Forts.)

operator* & operator->

```
const_reference operator*() {
  return(m_knot->m_data);
}

//*********

const_pointer operator->() {
  return(&(m_knot->m_data));
}
```

Listing 230: Die Methoden operator und operator-> von const_levelorder_iterator*

Vergleichsoperatoren

```
bool operator==(const_levelorder_iterator &i) const {
  return(m_knot==i.m_knot);
}

//*********

bool operator!=(const_levelorder_iterator &i) const {
  return(m_knot!=i.m_knot);
}
```

Listing 231: Die Vergleichsoperatoren von const_levelorder_iterator

operator++

```
const_levelorder_iterator &operator++() {
  if(m_knot->m_left)
    m_queue.push_back(m_knot->m_left);
  if(m_knot->m_right)
    m_queue.push_back(m_knot->m_right);
  if(m_queue.size()!=0) {
    m_knot=m_queue.front();
    m_queue.pop_front();
    if(!m_knot) {
      m_level++;
      m_queue.push_back(0);
      m_knot=m_queue.front();
      m_queue.pop_front();
    }
  }
  else
    m_knot=0;
  return(*this);
}

//*********

const_levelorder_iterator operator++(int) {
  const_levelorder_iterator tmp=*this;
  ++(*this);
  return(tmp);
}
```

Listing 232: Die Methoden operator++ von const_levelorder_iterator

getLevel

```
int getLevel() const {
  return(m_level);
}
```

Listing 233: Die getLevel-Methode von const_levelorder_iterator

Iterator-Methoden

```
const_levelorder_iterator levelorder_begin() const {
  return(const_levelorder_iterator(m_root,this));
}

//*********

const_levelorder_iterator levelorder_end() const {
  return(const_levelorder_iterator(0,this));
}
```

Listing 234: Die Iterator-Methoden für const_levelorder_iterator

Sie finden die Iteratoren eingebettet in die Klasse `CTree` auf der CD in der `CTree.h`-Datei.

43 Wie kann ein höhenbalancierter Binärbaum implementiert werden?

Wir haben in den letzten Rezepten einen Binärbaum implementiert und ihn mit Iteratoren bestückt. Was aber, wenn wir die Iteratoren auch auf einem ausgeglichenen Baum verwenden wollen?

Wir werden dazu einen ausgeglichenen Baum programmieren, der die wesentliche Funktionalität von `CTree` aus Rezept 38 übernimmt.

Von den verschiedenen Möglichkeiten, einen Baum auszugleichen, wollen wir den AVL-Baum wählen. Bei einem AVL-Baum besitzt jeder Knoten eine so genannte Balance. Die Balance ist definiert als die Differenz der Höhen des rechten und linken Teilbaums.

In einem AVL-Baum besitzt jeder Knoten eine Balance von –1, 0 oder 1.

Sollte diese Balance größer als 1 beziehungsweise kleiner als –1 werden, so ist die AVL-Bedingung verletzt und der Baum wird durch Rotationen wieder in einen AVL-Baum umgewandelt.

Auf die hinter diesen Mechanismen liegende Theorie kann hier leider nicht eingegangen werden.

CAVLKnot

In einem AVL-Baum besitzt jeder Knoten zusätzlich zu den Verweisen auf Vater, rechten und linken Sohn noch eine Balance. Wir leiten dazu die Klasse `CAVLKnot` von der Klasse `CKnot` aus `CTree` ab:

Klassendefinition

```
typedef short balance_type;

class CAVLKnot : public CAVLTree::CKnot {
public:
  balance_type m_balance;
};
```

Listing 235: Die Klassendefinition von CAVLKnot

Konstruktoren

Alle Konstruktoren benutzen zur Initialisierung der Knoten-Verweise und der Nutzdaten den Basisklassen-Konstruktor:

```
CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re)
: CKnot(fa,li,re),m_balance(0)
{ }

//*********

CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re,
         const value_type &v)
: CKnot(fa,li,re,v),m_balance(0)
{}

//*********

CAVLKnot(CAVLKnot *fa, CAVLKnot *li, CAVLKnot *re,
         const value_type &v, balance_type b)
: CKnot(fa,li,re,v),m_balance(b)
{}
```

Listing 236: Die Konstruktoren von CAVLKnot

Destruktor

Der Destruktor macht eigentlich nichts außer mit seiner Virtualität dafür Sorge zu tragen, dass der Destruktor einer eventuellen Unterklasse ebenfalls aufgerufen wird.

```
virtual ~CAVLKnot()
{}
```

Listing 237: Der Destruktor von CAVLKnot

Zugriffs-Methoden

Um innerhalb der Klasse `CAVLTree` die Problematik etwas zu verringern, dass die Knotenverweise eigentlich vom Typ `CKnot` sind, fügen wir drei Zugriffs-Methoden hinzu, die die Umwandlung übernehmen. Wir entscheiden uns hier für die ungeprüfte Typumwandlung, weil in unserem Kontext nur Knoten vom Typ `CAVLKnot` vorkommen können.

```
CAVLKnot *getLeft() const {
  return(reinterpret_cast<CAVLKnot*>(m_left));
}

//*********

CAVLKnot *getRight() const {
  return(reinterpret_cast<CAVLKnot*>(m_right));
}

//*********

CAVLKnot *getFather() const {
  return(reinterpret_cast<CAVLKnot*>(m_father));
}
```

Listing 238: Die Zugriffs-Methoden von CAVLKnot

CAVLTree

Damit im AVL-Baum ebenfalls die Iteratoren von `CTree` zur Verfügung stehen, leiten wir `CAVLTree` von `CTree` ab.

Klassendefinition

Genau wie `CTree` wird das Verhalten des AVL-Baums von Tree-Traits abhängig gemacht.

```
template<typename Traits>
class CAVLTree : public CTree<Traits> {
public:
  typedef CTree<Traits> base_type;
  typedef CAVLTree<Traits> my_type;
  typedef typename Traits::value_type value_type;
  typedef typename Traits::key_type key_type;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef short balance_type;

  class CAVLKnot : public CAVLTree::CKnot { /* ... */ };
};
```

Listing 239: Die Klassendefinition von CAVLTree

Konstruktoren

Genau wie `CTree` statten wir den AVL-Baum mit einem Standard-Konstruktor, einem Kopier-Konstruktor und einem Konstruktor aus, der den Baum mit Elementen aus einem mit Iteratoren definierten Bereich initialisiert:

```
CAVLTree(void)
: base_type() {
}

//*********

CAVLTree(const CAVLTree &t)
: base_type() {
  *this=t;
}

//*********
```

Listing 240: Die Konstruktoren von CAVLTree

```
template<typename Input>
CAVLTree(Input beg, Input end)
: base_type() {
  insert(beg, end);
}
```

Listing 240: Die Konstruktoren von CAVLTree (Forts.)

Destruktor

Der Destruktor muss keine Arbeit übernehmen, weil alle notwendigen Löschoperationen vom Destruktor der Basisklasse ausgeführt werden.

```
virtual ~CAVLTree()
{}
```

Listing 241: Der Destruktor von CAVLTree

Zuweisungsoperator

Der Zuweisungsoperator kopiert einen Baum knotenweise unter Zuhilfenahme von `copySons`.

```
CAVLTree &operator=(const CAVLTree &t) {
  if(&t!=this) {

/*
** Alten Baum löschen
*/
    deleteKnot(m_root);
    m_size=t.m_size;

/*
** Zuzuweisender Baum nicht leer?
** => alle Knoten kopieren
*/
    if(t.m_root) {

/*
```

Listing 242: Der Zuweisungsoperator von CAVLKnot

```
** Zunächst Wurzel kopieren
*/
      m_root=new CAVLKnot(0,0,0,t.m_root->m_data,
                  reinterpret_cast<CAVLKnot*>(t.m_root)->m_balance);

/*
** Söhne der Wurzel mit copySons kopieren
*/
      copySons(reinterpret_cast<CAVLKnot*>(m_root),
               reinterpret_cast<CAVLKnot*>(t.m_root));
    }
  }
  return(*this);
}
```

Listing 242: Der Zuweisungsoperator von CAVLKnot (Forts.)

copySons

Die Methode `copySons` kopiert die Söhne eines Knotens:

```
void copySons(CAVLKnot* d, CAVLKnot *s) {

/*
** Beide Knoten existent?
*/
  if(d&&s) {

/*
** Linken Sohn kopieren, falls vorhanden, und
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_left) {
      d->m_left=new CAVLKnot(d,0,0,s->m_left->m_data,
         reinterpret_cast<CAVLKnot*>(s->m_left)->m_balance);
      copySons(reinterpret_cast<CAVLKnot*>(d->m_left),
               reinterpret_cast<CAVLKnot*>(s->m_left));
    }

/*
** Rechten Sohn kopieren, falls vorhanden, und
```

Listing 243: Die Methode copySons von CAVLKnot

```
** für ihn copySons rekursiv aufrufen
*/
    if(s->m_right) {
      d->m_right=new CAVLKnot(d,0,0,s->m_right->m_data,
         reinterpret_cast<CAVLKnot*>(s->m_right)->m_balance);
      copySons(reinterpret_cast<CAVLKnot*>(d->m_right),
               reinterpret_cast<CAVLKnot*>(s->m_right));
    }
  }
}
```

Listing 243: Die Methode copySons von CAVLKnot (Forts.)

insert

Damit die beim Einfügen unter Umständen notwendigen Umstrukturierungen ausgeführt werden können, müssen in `CAVLTree` alle `insert`-Methoden der Basis-Klasse überschrieben werden.

Die erste `insert`-Methode erzeugt aus einem Nutzdaten-Objekt einen Knoten und ruft die dafür verantwortliche `insert`-Methode auf:

```
virtual inorder_iterator insert(const value_type &v) {

/*
** Datenobjekt in einen Knoten packen
*/
  CAVLKnot *kn=new CAVLKnot(0,0,0,v);

/*
** Knoten in Baum einfügen
*/
  insert(kn);

/*
** Position als Iterator zurückgeben
*/
  return(_iterator(kn,this,0));
}
```

Listing 244: Die Methode insert für value_type-Objekte

Um eine größere Kompatibilität zu anderen STL-Containern zu erreichen, fügen wir eine `insert`-Methode mit der zusätzlichen Angabemöglichkeit einer Einfüge-Position bei. Diese Position muss natürlich in einem Baum ignoriert werden:

```
inorder_iterator insert(_iterator i, const value_type &v) {
  return(insert(v));
}
```

Listing 245: Die Methode insert mit Einfüge-Position

Als letzte der öffentlichen `insert`-Methoden fehlt noch die Version, die einen mit Iteratoren definierten Bereich in den Baum einfügt:

```
template<typename Input>
void insert(Input beg, Input end) {
  while(beg!=end)
    insert(*(beg++));
}
```

Listing 246: Die insert-Methode für Bereiche

Zum Schluss fehlt noch die Methode, die den mit den anderen `insert`-Methoden erzeugten Knoten in den Baum einfügt.

```
void insert(CAVLKnot *kn) {

/*
** Baum leer?
** => Einzufügender Knoten wird die Wurzel
*/
  if(!m_root) {
    m_root=kn;
    m_size++;
    return;
  }

  CAVLKnot *cur=reinterpret_cast<CAVLKnot*>(m_root);
  while(cur) {
```

Listing 247: Die insert-Methode für Knoten

```
    if(Traits::gt(Traits::getKey(cur->m_data),
                  Traits::getKey(kn->m_data))) {

/*
** Einzufügender Knoten kleiner als aktueller Knoten?
** => Falls vorhanden, im linken Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten linker
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_left) {
        cur->m_left=kn;
        cur->m_balance--;
        kn->m_father=cur;
        m_size++;

/*
** Hat sich Balance von 0 auf anderen Wert geändert
** => Höhe hat sich geändert, eventuell umstrukturieren
*/
        if(cur->m_balance)
          rebalanceInsert(cur);
        return;
      }
      else {
        cur=cur->getLeft();
      }
    }
    else {

/*
** Einzufügender Knoten größer/gleich dem aktuellen Knoten?
** => Falls vorhanden, im rechten Teilbaum nach Einfügeposition
**    suchen. Andernfalls wird einzufügender Knoten rechter
**    Sohn des aktuellen Knotens
*/
      if(!cur->m_right) {
        cur->m_right=kn;
        cur->m_balance++;
        kn->m_father=cur;
        m_size++;
```

Listing 247: Die insert-Methode für Knoten (Forts.)

```
/*
** Hat sich Balance von 0 auf anderen Wert geändert
** => Höhe hat sich geändert, eventuell umstrukturieren
*/
        if(cur->m_balance)
          rebalanceInsert(cur);
        return;
      }
      else {
        cur=cur->getRight();
      }
    }
  }
}
```

Listing 247: Die insert-Methode für Knoten (Forts.)

rebalanceInsert

Die Methode `rebalanceInsert` steigt vom eingefügten Knoten aus den Baum nach oben und nimmt mit Hilfe der Methode `rotate` eventuell notwendige Korrekturen der Baumstruktur vor:

```
void rebalanceInsert(CAVLKnot *kn) {
  CAVLKnot *fkn=kn->getFather();

  if(((kn->m_balance==-1)||(kn->m_balance==1))&&(kn!=m_root)) {
    if(kn->m_father->m_left==kn)
      kn->getFather()->m_balance--;
    else
      kn->getFather()->m_balance++;
    rebalanceInsert(kn->getFather());
    return;
  }

  if(kn->m_balance==-2) {
    if(kn->getLeft()->m_balance==-1) {
      rotate(kn);
      return;
    }
    else {
```

Listing 248: Die Methode rebalanceInsert von CAVLTree

```
      rotate(kn->getLeft());
      rotate(kn);
      return;
    }
  }

   if(kn->m_balance==2) {
    if(kn->getRight()->m_balance==1) {
      rotate(kn);
      return;
    }
    else {
      rotate(kn->getRight());
      rotate(kn);
      return;
    }
  }
}
```

Listing 248: Die Methode rebalanceInsert von CAVLTree (Forts.)

erase

Die folgende `erase`-Methode löscht einen Knoten über seinen Schlüssel. Dabei werden alle Knoten mit diesem Schlüssel gelöscht und die Anzahl der gelöschten Knoten zurückgegeben.

```
virtual size_type erase(const key_type &k) {

/*
** Ersten Knoten mit Schlüssel k finden
*/
  CAVLKnot *kn=reinterpret_cast<CAVLKnot*>(findFirstKnot(k));

/*
** Keine Knoten vorhanden?
** => Kein Knoten gelöscht
*/
  if(!kn)
    return(0);
```

Listing 249: Die erase-Methode für key_type-Objekte

```
/*
** Inorder-Iterator auf Knoten hinter dem zu löschenden
** Knoten setzen
*/
  inorder_iterator i=_iterator(kn,this,0);
  ++i;
  size_type a=0;

/*
** So lange laufen, wie noch Knoten mit
** Schlüssel k existieren
*/
  do {

/*
** Schlüssel löschen, neuen Knoten aus Iterator
** holen und Iterator inkrementieren
*/
    erase(kn);
    a++;
    kn=reinterpret_cast<CAVLKnot*>(i.m_knot);
    ++i;
  } while((kn)&&(Traits::eq(Traits::getKey(kn->m_data),k)));
  return(a);
}
```

Listing 249: Die erase-Methode für key_type-Objekte (Forts.)

Die nachstehende Methode löscht den Knoten an der Iterator-Position `i` und liefert den nächsten Knoten der Inorder-Reihenfolge als Iterator zurück:

```
virtual inorder_iterator erase(_iterator i) {

/*
** Iterator-Position hinter zu löschendem
** Knoten ermitteln
** (Ohne Anfangs-Position für Iterator zu bestimmen)
*/
  inorder_iterator io(i.m_knot,this,0);
  ++io;
```

Listing 250: Die erase-Methode für Iterator-Positionen

```
/*
** Konnte Knoten gelöscht werden?
** => Position hinter gelöschtem Knoten zurückgeben
** Andernfalls End-Position zurückgeben
*/
  if(erase(reinterpret_cast<CAVLKnot*>(i.m_knot)))
    return(_iterator(io.m_knot,this,0));
  else
    return(inorder_end());
}
```

Listing 250: Die erase-Methode für Iterator-Positionen (Forts.)

Nun muss von der `CTree`-Klasse noch die `erase`-Methode für zu löschende Bereiche überschrieben werden.

Der Methode werden in Form von Iterator-Positionen der Beginn des zu löschenden Bereichs und die Position hinter dem letzten Element des zu löschenden Bereichs übergeben. Die Methode liefert die Position hinter dem gelöschten Bereich (bezogen auf die Inorder-Reihenfolge) zurück.

```
virtual inorder_iterator erase(inorder_iterator beg,
                               inorder_iterator end) {
  while((beg!=inorder_end())&&(beg!=end)) {
    CKnot *kn=beg.m_knot;
    ++beg;
    erase(reinterpret_cast<CAVLKnot*>(kn));
  }
  return(beg);
}
```

Listing 251: Die Methode erase für zu löschende Bereiche

Um einen Knoten aus dem Baum entfernen zu können, existiert die folgende `erase`-Methode:

```
bool erase(CAVLKnot *cur) {
  if(!cur) return(false);
```

Listing 252: Die erase-Methode für Knoten

```
  CAVLKnot *father=cur->getFather();

/*
** Zu löschender Knoten hat keine Söhne?
** => Kann problemlos gelöscht werden
*/
  if((!cur->m_left)&&(!cur->m_right)) {

/*
** Zu löschender Knoten ist die Wurzel?
** => Baum leer
*/
    if(cur==m_root) {
      m_root=0;
      delete(cur);
      m_size--;
      return(false);
    }

/*
** Zu löschender Knoten ist nicht die Wurzel?
** => Vater muss berücksichtigt werden
*/
    else {

/*
** Je nachdem, ob zu löschender Knoten der linke oder
** rechte Sohn des Vaters ist, muss entsprechender Sohn
** des Vaters auf 0 gesetzt werden
*/
      if(father->m_left==cur) {
        father->m_left=0;
        father->m_balance++;
      }
      else {
        father->m_right=0;
        father->m_balance--;
      }

/*
```

Listing 252: Die erase-Methode für Knoten (Forts.)

```
** Knoten löschen, AVL-Bedingung testen und
** ggfs. wieder herstellen
*/
      delete(cur);
      m_size--;
      rebalanceErase(father);
      return(true);
    }
  }

/*
** Besitzt zu löschender Knoten zwei Söhne?
** => Zu löschenden Knoten durch symmetrischen
**    Vorgänger ersetzen und symmetrischen Vorgänger
**    löschen
*/
  if((cur->m_left)&&(cur->m_right)) {
    CAVLKnot *sys=reinterpret_cast<CAVLKnot*>(symmetricPred(cur));
    cur->m_data=sys->m_data;
    return(erase(sys));
  }

/*
** Besitzt zu löschender Knoten nur einen Sohn?
** => Sohn des zu löschenden Knotens wird Sohn vom
** Vater des zu löschenden Knotens
*/
  CAVLKnot *son;
  if(cur->m_left)
    son=cur->getLeft();
  else
    son=cur->getRight();

  if(cur!=m_root) {
    son->m_father=father;
    if(father->m_left==cur) {
      father->m_left=son;
      father->m_balance++;
    }
    else {
      father->m_right=son;
```

Listing 252: Die erase-Methode für Knoten (Forts.)

```
      father->m_balance--;
    }

/*
** AVL-Bedingung testen und ggfs. wieder herstellen
*/
    rebalanceErase(father);
    }

/*
** Ist zu löschender Sohn die Wurzel?
** => Sohn des zu löschenden Knotens wird Wurzel
*/
  else {
    son->m_father=0;
    m_root=son;
  }

/*
** Knoten löschen
*/
  delete(cur);
  m_size--;
  return(true);
}
```

Listing 252: Die erase-Methode für Knoten (Forts.)

rebalanceErase

`rebalanceErase` stellt eine eventuell verletzte AVL-Bedingung nach dem Entfernen eines Knotens wieder her.

```
void rebalanceErase(CAVLKnot *kn) {
  CAVLKnot *fkn=kn->getFather();

  if((kn->m_balance==-1)||(kn->m_balance==1))
    return;
  if((kn==m_root)&&(kn->m_balance==0))
    return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree

```
  if(kn==m_root) {
    if(kn->m_balance==-2) {
      if(kn->getLeft()->m_balance<=0)
        rotate(kn);
      else {
        kn=rotate(kn->getLeft());
        rotate(kn);
      }
    }
    else {
      if(kn->getRight()->m_balance>=0)
        rotate(kn);
      else {
        kn=rotate(kn->getRight());
        rotate(kn);
      }
    }
    return;
  }
  if(kn->m_balance==2) {
    switch(kn->getRight()->m_balance) {
      case 0:
        rotate(kn);
        return;

      case 1:
        rebalanceErase(rotate(kn));
        return;

      case -1:
        rotate(kn->getRight());
        rebalanceErase(rotate(kn));
        return;
    }
  }
  if(kn->m_balance==-2) {
    switch(kn->getLeft()->m_balance) {
      case 0:
        rotate(kn);
        return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

```
      case -1:
        rebalanceErase(rotate(kn));
        return;

      case 1:
        rotate(kn->getLeft());
        rebalanceErase(rotate(kn));
        return;
    }
  }
  if(fkn->m_left==kn) {
    fkn->m_balance++;
    if(fkn->m_balance<2) {
      rebalanceErase(fkn);
      return;
    }
    switch(fkn->getRight()->m_balance) {
      case 0:
        rotate(fkn);
        return;

      case 1:
        rebalanceErase(rotate(fkn));
        return;

      case -1:
        rotate(fkn->getRight());
        rebalanceErase(rotate(fkn));
        return;
    }
  }
  if(fkn->m_right==kn) {
    fkn->m_balance--;
    if(fkn->m_balance>-2) {
      rebalanceErase(fkn);
      return;
    }
    switch(fkn->getLeft()->m_balance) {
      case 0:
        rotate(fkn);
        return;
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

```
      case -1:
        rebalanceErase(rotate(fkn));
        return;

      case 1:
        rotate(fkn->getLeft());
        rebalanceErase(rotate(fkn));
        return;
    }
  }
}
```

Listing 253: Die Methode rebalanceErase von CAVLTree (Forts.)

rotate

Die Methode `rotate` wird von `rebalanceInsert` und `rebalanceErase` eingesetzt, um den Baum umzustrukturieren und ihn damit wieder zu einem AVL-Baum zu machen.

Genau wie `rebalanceInsert` und `rebalanceErase` wird `rotate` der Vollständigkeit halber aufgeführt, bleibt aber unkommentiert, weil eine Erklärung der dahinter liegenden Theorie hier den Rahmen sprengen würde.

```
CAVLKnot *rotate(CAVLKnot *kn) {
  CAVLKnot *child;

  if(kn->m_balance<0) {
    child=kn->getLeft();
    kn->m_left=child->m_right;
    if(child->m_right)
      child->m_right->m_father=kn;
    child->m_right=kn;
    child->m_father=kn->m_father;
    kn->m_father=child;
    if(child->m_father) {
      if(child->m_father->m_left==kn)
        child->m_father->m_left=child;
      else
        child->m_father->m_right=child;
```

Listing 254: Die Methode rotate von CAVLTree

```
    }
    else
      m_root=child;

    if(kn->m_balance==-1) {
      if(child->m_balance==1) {
        child->m_balance=2;
        kn->m_balance=0;
        return(child);
      }

      if(child->m_balance==-1)
        kn->m_balance=1;
      else
        kn->m_balance=0;
      child->m_balance=1;
      return(child);
    }
    if(kn->m_balance==-2) {
      if(child->m_balance==-1) {
        kn->m_balance=child->m_balance=0;
        return(child);
      }
      if(child->m_balance==0) {
        kn->m_balance=-1;
        child->m_balance=1;
        return(child);
      }
      if(child->m_balance==-2) {
        kn->m_balance=1;
        child->m_balance=0;
        return(child);
      }
    }
  }
  else {
    child=kn->getRight();
    kn->m_right=child->m_left;
    if(child->m_left) child->m_left->m_father=kn;
    child->m_left=kn;
    child->m_father=kn->m_father;
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

```
    kn->m_father=child;
    if(child->m_father) {
      if(child->m_father->m_left==kn)
        child->m_father->m_left=child;
      else
        child->m_father->m_right=child;
    }
    else
      m_root=child;

    if(kn->m_balance==1) {
      if(child->m_balance==-1) {
//            Inorder();
        child->m_balance=-2;
        kn->m_balance=0;
        return(child);;
      }

      if(child->m_balance==1)
        kn->m_balance=-1;
      else
        kn->m_balance=0;
      child->m_balance=-1;
      return(child);
    }
    if(kn->m_balance==2) {
      if(child->m_balance==1)  {
        kn->m_balance=child->m_balance=0;
        return(child);
      }
      if(child->m_balance==0)  {
        kn->m_balance=1;
        child->m_balance=-1;
        return(child);
      }
      if(child->m_balance==2)  {
        kn->m_balance=-1;
        child->m_balance=0;
        return(child);
      }
    }
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

```
  }
  return(child);
}
```

Listing 254: Die Methode rotate von CAVLTree (Forts.)

pop_front & pop_back

Mittels `pop_front` und `pop_back` können Knoten bezogen auf die Inorder-Reihenfolge am Anfang oder am Ende des Baumes entfernt werden.

```
virtual void pop_front() {
  if(!empty())
    erase(reinterpret_cast<CAVLKnot*>(inorder_begin().m_knot));
}

//*********

virtual void pop_back() {
  if(!empty())
    erase(reinterpret_cast<CAVLKnot*>(inorder_rbegin().m_knot));
}
```

Listing 255: Die Methoden pop_front und pop_back

push_front & push_back

Der Kompatibilität wegen dem Baum hinzugefügt, halten sie allerdings nicht, was ihr Name verspricht:

```
virtual void push_front(const value_type &v) {
  insert(v);
}

//*********

virtual void push_back(const value_type &v) {
  insert(v);
}
```

Listing 256: Die Methoden push_front und push_back

Sie finden die Klasse `CAVLTree` auf der CD in der *CAVLTree.h*-Datei.

CAVLSetTree

Um den Baum wie ein `set` der STL nutzen zu können, implementieren wir eine Klasse `CAVLSetTree` und benutzen dazu die `set_tree_traits` aus Rezept 38:

```
template<typename VType, typename Cmp= std::less<VType> >
class CAVLSetTree : public CAVLTree<set_tree_traits<VType, Cmp> > {
public:
  typedef CAVLTree<set_tree_traits<VType, Cmp> > base_type;
  typedef CAVLSetTree<VType, Cmp> my_type;

//*********

  CAVLSetTree()
  : base_type()
  {}

//*********

  template<typename Input>
  CAVLSetTree(Input beg, Input end)
  : base_type(beg,end)
  { }

//*********

  CAVLSetTree(const CAVLSetTree &t)
  : base_type(t)
  { }
};
```

Listing 257: Die Klasse CAVLSetTree

Sie finden die Klasse `CAVLSetTree` auf der CD in der *CAVLSetTree.h*-Datei.

CAVLMapTree

Analog zu `CAVLSetTree` stellen wir die `map`-Funktionalität mit der Klasse `CAVLMapTree` her und benutzen dazu die `map_tree_traits` aus Rezept 38:

```
template<typename KType,
         typename VType,
         typename Cmp= std::less<KType> >
class CAVLMapTree : public CAVLTree<map_tree_traits<KType,
                                                    VType,
                                                    Cmp> > {
public:
  typedef CAVLTree<map_tree_traits<KType, VType, Cmp> > base_type;
  typedef CAVLMapTree<KType, VType, Cmp> my_type;

//*********

  CAVLMapTree()
  : base_type()
  {}

//*********

  template<typename Input>
  CAVLMapTree(Input beg, Input end)
  : base_type(beg,end)
  { }

//*********

  CAVLMapTree(const CAVLMapTree &t)
  : base_type(t)
  { }
};
```

Listing 258: Die Klasse CAVLMapTree

Sie finden die Klasse `CAVLMapTree` auf der CD in der *CAVLMapTree.h*-Datei.

Datum & Uhrzeit

44 Wie wird auf Schaltjahr geprüft?

Bis zum Jahr 1582 war die Welt noch einfach, denn jedes durch vier teilbare Jahr war ein Schaltjahr (Julianischer Kalender).

Ab 1583 jedoch hat sich die Regel etwas verkompliziert (Gregorianischer Kalender): Ein durch vier teilbares Jahr ist nur dann ein Schaltjahr, wenn es nicht durch 100 teilbar ist. Und ein durch 100 teilbares Jahr ist nur dann kein Schaltjahr, wenn es nicht durch 400 teilbar ist.

isLeapYear

Die Funktion `isLeapyear` setzt diese Regeln um:

```
bool isLeapyear(int year) {
  if(year<1)
    throw out_of_range("isLeapYear");

  if(year%4) return(false);
  if(year<=1582) return(true);
  if(year%100) return(true);
  if(year%400) return(false);
  return(true);
}
```

Listing 259: Die Funktion isLeapyear

Sollte ein Jahr kleiner 1 übergeben werden, dann wird die Ausnahme `out_of_range` aus der Standardbibliothek geworfen.

Sie finden die Funktion als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien.

Das unten stehende Beispiel ermittelt die Schaltjahre im Bereich 2000-2400:

```
#include "CDate.h"
#include <iostream>
```

Listing 260: Ein Beispiel zu isLeapYear

```
using namespace std;
using namespace Codebook;

int main() {
  int count=0;
  for(int i=2000; i<=2400; i++)
    if(CDate::isLeapyear(i)) {
      cout << i;
      if(!(++count%10))
        cout << endl;
      else
        cout << ", ";
    }
  cout << endl;
}
```

Listing 260: Ein Beispiel zu isLeapYear (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0401.cpp*.

Die Ausgabe in Abbildung 29 zeigt sehr schön die Ausnahmefälle. Obwohl das Jahr 2100 durch 4 teilbar ist, ist es dennoch kein Schaltjahr. Und das Jahr 2400 ist ein Schaltjahr, trotzdem es durch 100 teilbar ist.

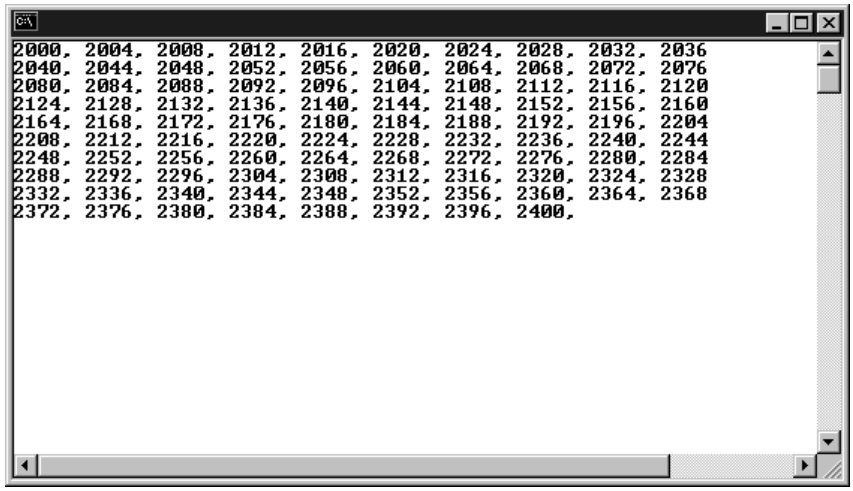


Abbildung 29: Die Schaltjahre von 2000 bis 2400

45 Wie viele Tage hat ein Monat?

Weil die Anzahl der Tage zumindest beim Februar abhängig vom Jahr ist (Schaltjahr oder nicht), muss der Funktion `daysPerMonth` sowohl der Monat als auch das gewünschte Jahr übergeben werden:

daysPerMonth

```
int daysPerMonth(int month, int year) {
  switch(month) {
  case 1: case 3: case 5: case 7:
  case 8: case 10: case 12:
    return(31);

  case 4: case 6: case 9: case 11:
    return(30);

  case 2: return(28+isLeapyear(year));

  default:
    throw out_of_range("daysPerMonth");
  }
}
```

Listing 261: Die Funktion daysPerMonth

Sie finden die Funktion als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien. `daysPerMonth` macht von der Funktion `isLeapyear` aus Rezept 44 Gebrauch.

46 Wie viele Tage sind seit dem 1.1.0001 vergangen?

Eine Antwort auf diese Frage scheint auf den ersten Blick nicht sonderlich interessant. Ein solcher Ansatz gibt aber die Möglichkeit, alle Daten auf einer linearen Achse zu sehen.

Eine besondere Rolle spielt hier das Jahr 1582. Bis zu diesem Jahr verschob sich das Datum wegen der Ungenauigkeit des Julianischen Kalenders um zehn Tage.

Deswegen ordnete der damalige Papst an, dass die zehn Tage vom 5.10.1582 bis zum 14.10.1582 kurzerhand gestrichen werden sollten. Auf Donnerstag, den 4.10.1582, folgte damit Freitag, der 15.10.1582.

calcDaysPast

Von dieser Korrektur an hatte der Julianische Kalender ausgedient und wurde durch den Gregorianischen Kalender ersetzt. Diese »Lücke« in der Zeitrechnung muss von unserer Funktion `calcDaysPast` berücksichtigt werden:

```
long calcDaysPast(int day, int month, int year) {

/*
** Grenzen überprüfen
*/
  if((day<1)||(month<1)||(year<1)||
     (month>12)||(day>daysPerMonth(month, year))||
     ((year==1582)&&(month==10)&&(day>=5)&&(day<=14)))
    throw out_of_range("calcDaysPast");

/*
** Invariante: Der 1.1.0001 ist Tag 0
*/
  long daysPast=0;
  int x=1;

/*
** Tage bis zum aktuellen Jahr addieren
*/
  while(x<year)
    daysPast+=365+isLeapyear(x++);
  x=1;

/*
** Tage bis zum aktuellen Monat addieren
*/
  while(x<month)
    daysPast+=daysPerMonth(x++,year);

/*
** Tage des aktuellen Monats addieren
*/
```

Listing 262: Die Funktion calcDaysPast

```
  daysPast+=day-1;

/*
** Lag Datum nach dem 14.10.1582?
** => 10 Tage abziehen wegen Lücke
*/
  if(daysPast>577736)
    daysPast-=10;
  return(daysPast);
}
```

Listing 262: Die Funktion calcDaysPast (Forts.)

Sie finden die Funktion als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien. `calcDaysPast` macht von den Funktionen `isLeapyear` aus Rezept 44 und `daysPerMonth` aus Rezept 45 Gebrauch.

47 Auf welchen Wochentag fällt ein bestimmtes Datum?

Wenn wir das Datum als Anzahl der Tage vorliegen haben, dann wissen wir, dass sieben Tage später (oder früher) derselbe Wochentag ist.

Unter der Voraussetzung, dass der 1.1.0001 ein Samstag ist (und das ist er), brauchen wir lediglich bis zum gewünschten Datum hochzurechnen und haben den Wochentag.

calcWeekDay

Diese Aufgabe erledigt `calcWeekday` unter Zuhilfenahme von `calcDaysPast` aus Rezept 46:

```
string calcWeekday(int day, int month, int year) {

/*
** Invariante: Der 1.1.0001 ist ein Samstag
*/
  switch((calcDaysPast(day,month,year)+5)%7) {
    case 0: return("Montag");
    case 1: return("Dienstag");
```

Listing 263: Die Funktion calcWeekday

```
    case 2: return("Mittwoch");
    case 3: return("Donnerstag");
    case 4: return("Freitag");
    case 5: return("Samstag");
    case 6: return("Sonntag");
    default: throw out_of_range("calcWeekday");

  }
}
```

Listing 263: Die Funktion calcWeekday (Forts.)

Sie finden die Funktion als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien. Sie macht von `calcDaysPast` aus Rezept 46 Gebrauch.

Das folgende Beispiel ermittelt den Wochentag, an dem das Deutsche Grundgesetz angenommen wurde (es war ein Montag):

```
#include "CDate.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  cout << "Das Deutsche Grundgesetz wurde an einem ";
  cout << CDate::calcWeekday(23,5,1949);
  cout << " angenommen." << endl;
}
```

Listing 264: Ein Beispiel zu calcWeekday

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0404.cpp*.

48 Wie kann ein Datum vernünftig repräsentiert werden?

Einen Ansatz für das Speichern eines Datums samt Uhrzeit bietet die Struktur `tm` aus der Header-Datei `ctime`:

```
struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};
```

Listing 265: Die Struktur tm aus ctime

Die einzelnen Elemente haben folgende Bedeutung:

- `tm_sec` - die Sekunden seit Beginn der aktuellen Minute (0-59)
- `tm_min` - die Minuten seit Beginn der aktuellen Stunde (0-59)
- `tm_hour` - die Stunden seit Mitternacht (0-23)
- `tm_mday` - der Tag im aktuellen Monat (1-31)
- `tm_mon` - der Monat seit Januar (0-11)
- `tm_year` - das Jahr seit 1900
- `tm_wday` - die Tage seit Sonntag (0-6)
- `tm_yday` - die Tage seit dem 1. Januar des aktuellen Jahres
- `tm_isdst` - Flag für Sommerzeit (0=Normalzeit, 1=Sommerzeit)

Eine Einschränkung fällt gleich bei `tm_year` auf: Daten vor 1900 können mit ihr nicht ausgedrückt werden.

Für ein Teminplaner-Programm mag das vielleicht noch keine wirkliche Einschränkung sein, aber für einen Ahnenforscher ist die Struktur geradezu unbrauchbar.

Die Funktionen aus den vorigen Rezepten besitzen als untere Grenze den 1.1.0001. Darauf basierend wollen wir eine Klasse `CDate` implementieren. Sie besitzt folgende Attribute:

- `m_day` – Tag (1-31)
- `m_month` – Monat (1-12)
- `m_year` – Jahr (1-32767)
- `m_daysPast` – Tage seit dem 1.1.0001

Zu diesen Attributen gibt es die üblichen `get`- und `set`-Methoden. Beim Setzen von Tag, Monat oder Jahr ist darauf zu achten, dass der Wert `m_daysPast` neu berechnet wird.

Die in den vorigen Rezepten zum Thema Datum besprochenen Funktionen wurden in `CDate` als statische Methoden aufgenommen. Sie werden an dieser Stelle jedoch nicht weiter besprochen. Bei Bedarf lesen Sie bitte im entsprechenden Rezept nach.

Klassendefinition

Hier zunächst einmal die Klassendefinition:

```
class CDate {
private:
  int m_day;
  int m_month;
  int m_year;
  long m_daysPast;

  void calcDaysPast();
  void calcDate();
  void checkRanges();

public:

  static bool isLeapyear(int year);
  static int daysPerMonth(int month, int year);
  static long calcDaysPast(int year, int month, int day);
  static std::string calcWeekday(int day, int month, int year);

  CDate(int day, int month, int year);
  CDate(long);
```

Listing 266: Die Definition der Klasse CDate

```
  CDate(const std::string &date);
  CDate();

  int getDay() const {
    return(m_day);
  }

  int getMonth() const {
    return(m_month);
  }

  int getYear() const {
    return(m_year);
  }

  long getDaysPast() const {
    return(m_daysPast);
  }

  void setDay(long w) {
    m_day=w;
    checkRanges();
    calcDaysPast();
  }

  void setMonth(long w) {
    m_month=w;
    checkRanges();
    calcDaysPast();
  }

  void setYear(long w) {
    m_year=w;
    checkRanges();
    calcDaysPast();
  }
};
```

Listing 266: Die Definition der Klasse CDate (Forts.)

Konstruktoren

Der Standard-Konstruktor von `CDate` wartet gleich mit einer Besonderheit auf: Er initialisiert das Objekt mit dem aktuellen Datum. Dazu bedient er sich der Funktionen `time` und `localtime` sowie der weiter oben besprochenen Struktur `tm`:

```
CDate::CDate() {

/*
** Aktuelles Datum und Uhrzeit holen
*/
  time_t t;
  time(&t);

/*
** Datum und Uhrzeit in tm-Struktur umwandeln
*/
  tm *ts;
  ts=localtime(&t);

/*
** Benötigte Daten aus Struktur auslesen
*/
  m_day=ts->tm_mday;
  m_month=ts->tm_mon+1;
  m_year=ts->tm_year+1900;

  checkRanges();
  calcDaysPast();
}
```

Listing 267: Der Standard-Konstruktor

Der Konstruktor macht Gebrauch von den Methoden `calcDaysPast` (Beschreibung in Rezept 46) und `checkRanges`.

checkRanges

`checkRanges` prüft die Attribute `m_day`, `m_month` und `m_year` auf gültige Werte. Dabei wird auch darauf geachtet, dass kein Datum in der in Rezept 46 beschriebenen Datumslücke liegt:

```
void CDate::checkRanges() {
  if((m_year<1)||(m_year>32767))
    throw out_of_range("CDate::checkRanges");
  if((m_month<1)||(m_month>12))
    throw out_of_range("CDate::checkRanges");
  if((m_day<1)||(m_day>daysPerMonth(m_month,m_year)))
    throw out_of_range("CDate::checkRanges");

  if((m_year==1582)&&(m_month==10)&&(m_day>4)&&(m_day<15))
    throw out_of_range("CDate::checkRanges");
}
```

Listing 268: Die Methode checkRanges

Dann bekommt die Klasse noch einen Konstruktor für die explizite Angabe von Tag, Monat und Jahr:

```
CDate::CDate(int day, int month, int year)
: m_day(day), m_month(month), m_year(year) {

  checkRanges();
  calcDaysPast();
}
```

Listing 269: Konstruktor für Tag, Monat und Jahr

Der nächste Konstruktor bekommt lediglich die seit dem 1.1.0001 vergangenen Tage übergeben und berechnet daraus das Datum:

```
CDate::CDate(long days)
: m_daysPast(days){
  if(days<0)
    throw out_of_range("CDate::CDate(long)");

  calcDate();
  checkRanges();
}
```

Listing 270: Konstruktor für die seit 1.1.0001 vergangenen Tage

Und der letzte Konstruktor extrahiert das Datum aus einem String der Form »t.m.j«, »t-mmm-j« oder »t mmm j«, wobei mmm eine dreistellige englische Abkürzung des Monats ist.

Tage und numerische Angaben des Monats können ein- oder zweistellig sein.

Das Jahr darf zwei- oder vierstellig sein. Bei einer zweistelligen Jahresangabe werden die ersten beiden Ziffern des Jahres vom aktuellen Datum genommen:

```
CDate::CDate(const string &date) {

/*
** Abkürzungen für Monate festlegen
*/
  static const char* const months[]={"Jan","Feb", "Mar",
                                     "Apr", "May", "Jun",
                                     "Jul", "Aug", "Sep",
                                     "Oct", "Nov", "Dec"};

/*
** Datum in Einzelteile zerlegen
** Es müssen genau drei Einzelteile sein.
*/
  vector<string> components=chopIntoWords(date," .-");
  if(components.size()!=3)
    throw invalid_argument("CDate::CDate(string)");

/*
** Tag umwandeln
*/
  m_day=atol(components[0].c_str());

/*
** Jahr umwandeln
** Bei zweistelligem Jahr die ersten beiden Ziffern
** des aktuellen Jahres nehmen
*/
  if(components[2].size()==4)
    m_year=atol(components[2].c_str());
  else if(components[2].size()==2)
    m_year=atol(components[2].c_str())+
           (CDate().getYear()/100*100);
```

Listing 271: Konstruktor für Datum im String-Format

```
  else
    throw invalid_argument("CDate::CDate(string)");

/*
** Datumskomponenten durch Punkte getrennt?
** => Monat ist als numerischer Wert angegeben
*/
  if(date.find(".")!=string::npos) {
    m_month=atol(components[1].c_str());
  }

/*
** Datumskomponenten durch Leerzeichen getrennt?
** => Monat ist als Abkürzung angegeben
*/
  else if((date.find(" ")!=string::npos)||
          (date.find("-")!=string::npos)) {
    insensitive_string m=components[1].c_str();
    for(int i=0; (i<12)&&(m!=months[i]); i++){}
    m_month=i+1;
  }
  else
    throw invalid_argument("CDate::CDate(string)");

  checkRanges();
  calcDaysPast();
}
```

Listing 271: Konstruktor für Datum im String-Format (Forts.)

calcDate

Die tatsächliche Berechnung des Datums aus `m_daysPast` ist in die private Methode `calcDate` ausgelagert. Sie ist gewissermaßen die Umkehrfunktion zu `calcDaysPast`:

```
void CDate::calcDate() {
  long days=m_daysPast;

/*
** Datum nach 4.10.1582?
** => 10 Tage der Datumslücke aufaddieren
```

Listing 272: Die private Methode calcDate

```
*/
  if(days>577736)
    days+=10;

/*
** Jahre von den Tagen abziehen, so lange
** noch ganze Jahre übrig sind
*/
  m_year=1;
  while(((days>=365)&&(!isLeapyear(m_year)))||
        ((days>=366)&&(isLeapyear(m_year)))) {
    days-=365+isLeapyear(m_year);
    m_year++;
  }

/*
** Monate von den Tagen abziehen, so lange
** noch ganze Monate übrig sind
*/
  m_month=1;
  while(days>=daysPerMonth(m_month,m_year)) {
    days-=daysPerMonth(m_month,m_year);
    m_month++;
  }

/*
** Übrig gebliebene Tag in m_day speichern
*/
  m_day=1+days;

  checkRanges();
}
```

Listing 272: Die private Methode calcDate (Forts.)

getDDMMYYYY

Um das Datum in einer darstellbaren Form zu haben, wurde die Methode `getDDMMYYYY` implementiert, die das Datum als String der Form »01.01.0001« zurückgibt:

```
string CDate::getDDMMYYYY() const {

  stringstream str;

  str.fill('0');
  str << setw(2) << m_day << ".";
  str << setw(2) << m_month << ".";
  str << setw(4) << m_year;
  return(str.str());
}
```

Listing 273: Die Methode getDDMMYYYY

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`:

Ein Objekt der Klasse `CDate` ist komplett durch den Wert `m_daysPast` dargestellt. Alle anderen Attribute lassen sich daraus errechnen. Es muss jedoch darauf geachtet werden, dass diese Berechnung beim Laden durchgeführt wird.

Die Stream-Operatoren müssen als Freunde der Klasse deklariert werden.

```
CBinaryOStream &operator<<(CBinaryOStream &os, const CDate &d) {
  os << d.m_daysPast;
  return(os);
}

CBinaryIStream &operator>>(CBinaryIStream &is, CDate &d) {
  is >> d.m_daysPast;
  d.calcDate();
  d.checkRanges();
  return(is);
}
```

Listing 274: Die Stream-Operatoren für binäre Ströme

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

49 Wie können die Vergleiche »früher« und »später« für Daten implementiert werden?

Wenn wir die Klasse `CDate` aus Rezept 48 als Grundlage nehmen, die Daten auf einer linearen Zeitlinie verwaltet, ist das Problem im Handumdrehen gelöst:

Ein Datum `d1` ist auf der Zeitlinie genau dann früher als ein Datum `d2`, wenn `d1` weniger Tage vom 1.1.0001 entfernt ist als `d2`.

Die Vergleichsoperatoren von `CDate` müssen also nur das Attribut `m_daysPast` vergleichen:

```
bool CDate::operator<(const CDate &d) const {
  return(m_daysPast<d.m_daysPast);
}

//*********

bool CDate::operator==(const CDate &d) const {
  return(m_daysPast==d.m_daysPast);
}
```

Listing 275: Die Vergleichsoperatoren == und < für CDate

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

50 Wie werden Zeitabstände zwischen Daten berechnet?

Auch dieses Problem löst sich mit der `CDate`-Klasse aus Rezept 48 nahezu selbstständig.

Wir brauchen lediglich die Differenz der beiden `m_daysPast`-Attribute zu bilden. Tage können nicht negativ werden, deswegen wird der Absolutwert der Differenz zurückgegeben:

```
long CDate::operator-(const CDate &d) const {
  return(labs(m_daysPast-d.m_daysPast));
}
```

Listing 276: CDate::operator- für CDate-Objekte

Es sollte auch möglich sein, von einem Datum Tage abziehen bzw. auf ein Datum Tage aufaddieren zu können. Dazu implementieren wir zwei weitere `operator`-Methoden:

```
CDate CDate::operator+(long days) const {
  return(CDate(m_daysPast+days));
}

//*********

CDate CDate::operator-(long days) const {
  return(CDate(m_daysPast-days));
}
```

Listing 277: Die CDate-Operatoren + und – für Tage

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

51 Auf welches Datum fällt der Ostersonntag?

Ostern zählt zu den beweglichen Feiertagen. Per Definition ist der Ostersonntag der erste Sonntag nach dem ersten Vollmond im Frühling.

Basierend auf einer Formel, die von Jean Meeus veröffentlicht wurde und wahrscheinlich auf Karl Friedrich Gauss und Jean Baptiste Joseph Delambre zurückzuführen ist, berechnet `calcEasterSunday` für jedes beliebige Jahr das Datum des Ostersonntags.

calcEasterSunday

Weil meine astronomischen Kenntnisse mit diesen Berechnungen hoffnungslos überfordert sind, bleibt der Quellcode unkommentiert:

```
CDate CDate::calcEasterSunday(int year) {
  if(year<1)
    throw(out_of_range("CDate::calcEasterSunday"));
  if(year>1582) {
    int A = year % 19;
    int B = year / 100;
    int C = year % 100;
```

Listing 278: Die Methode calcEasterSunday

```
    int D = B / 4;
    int E = B % 4;
    int F = (B + 8) / 25;
    int G = (B - F + 1) / 3;
    int H = (19*A + B - D - G + 15) % 30;
    int I = C / 4;
    int K = C % 4;
    int L = (32 + 2*E + 2*I - H - K) % 7;
    int M = (A + 11*H + 22*L) / 451;
    int P = (H + L - 7*M + 114) / 31;
    int Q = (H + L - 7*M + 114) % 31;
    return(CDate(Q+1,P,year));
  }
  else {
    int A = year % 4;
    int B = year % 7;
    int C = year % 19;
    int D = (19*C + 15) % 30;
    int E = (2*A + 4*B - D + 34) % 7;
    int F = (D + E + 114) / 31;
    int G = (D + E + 114) % 31;
    return(CDate(G+1,F,year));
  }
}
```

Listing 278: Die Methode calcEasterSunday (Forts.)

Sie finden die Funktion als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien.

Das nachfolgende Beispiel berechnet von 2000 bis 2010 die Karfreitage:

```
#include "CDate.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  for(int i=2000; i<=2010; i++) {
    cout << "Im Jahr " << i << " liegt der Karfreitag ";
```

Listing 279: Ein Beispiel zu calcEasterSunday

```
    cout << "auf dem ";
    cout << (CDate::calcEasterSunday(i)-2).getDDMM() << endl;
  }
}
```

Listing 279: Ein Beispiel zu calcEasterSunday (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0408.cpp*.

52 Wie werden andere, von Ostern abhängige Feiertage berechnet?

Es gibt noch andere bewegliche Feiertage, viele von ihnen richten sich nach Ostern. Im Folgenden sind diese Feiertage aufgeführt:

- Weiberfastnacht, ein wichtiger Feiertag der Rheinländer, fällt auf den Donnerstag 52 Tage vor Ostern.
- Aschermittwoch fällt auf den Mittwoch 46 Tage vor Ostern.
- Christi Himmelfahrt fällt auf den Donnerstag 39 Tage nach Ostern.
- Zehn Tage später, also 49 Tage nach Ostern, ist Pfingsten.
- Der letzte von Ostern abhängige Feiertag, Fronleichnam, fällt auf den Donnerstag 60 Tage nach Ostern.

Die dazugehörigen statischen `CDate`-Methoden sehen so aus.

```
/*
** Aschermittwoch
*/
CDate CDate::calcAshWednesday(int year) {
  return(calcEasterSunday(year)-46);
}

/*
** Weiberfastnacht
*/

CDate CDate::calcWeiberfastnacht(int year) {
```

Listing 280: Berechnung ostern-abhängiger Feiertage

```
  return(calcEasterSunday(year)-52);
}

/*
** Pfingsten
*/
CDate CDate::calcWhitsun(int year) {
  return(calcEasterSunday(year)+49);
}
/*
** Christi Himmelfahrt
*/
CDate CDate::calcAscensionDay(int year) {
  return(calcEasterSunday(year)+39);
}

/*
** Fronleichnam
*/

CDate CDate::calcCorpusChristi(int year) {
  return(calcEasterSunday(year)+60);
}
```

Listing 280: Berechnung ostern-abhängiger Feiertage (Forts.)

Sie finden die Funktionen als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien.

53 Auf welchen Tag fällt Muttertag?

Muttertag fällt auf den zweiten Sonntag im Mai. Sollte an diesem Sonntag Pfingsten liegen, dann wird Muttertag eine Woche vorverlegt.

Für diese Berechnung wird die `CDate`-Klasse aus dem vorigen Rezept um die Methode `calcMothersDay` erweitert.

Um den zweiten Sonntag bestimmen zu können, benötigen wir eine `calcWeekday`-Methode, die uns den Wochentag nicht als String, sondern als numerischen Wert liefert.

getWeekday & getWeekdayName

Wir spalten dazu die Funktionalität der statischen Methoden `calcWeekday` in die Methoden `getWeekday` und `getWeekdayName` auf:

```
int CDate::getWeekday() {

/*
** Invariante: Der 1.1.0001 ist ein Samstag
*/
  return((m_daysPast+5)%7);
}

string CDate::getWeekdayName() {

/*
** Invariante: Der 1.1.0001 ist ein Samstag
*/
  switch(getWeekday()) {
    case 0: return("Montag");
    case 1: return("Dienstag");
    case 2: return("Mittwoch");
    case 3: return("Donnerstag");
    case 4: return("Freitag");
    case 5: return("Samstag");
    case 6: return("Sonntag");
    default: throw out_of_range("getWeekdayName");
  }
}
```

Listing 281: Die Methoden calcWeekday und calcWeekdayName

calcMothersDay

Um den Muttertag zu bestimmen, wird zunächst der Wochentag des 1.5. im gewünschten Jahr ermittelt. Damit kann dann der zweite Sonntag im Mai berechnet werden.

Sollte dieser auf Pfingsten fallen, wird der Muttertag um eine Woche (sieben Tage) vorverlegt:

```
CDate CDate::calcMothersDay(int year) {

/*
** Zweiten Sonntag im Mai ermitteln
*/
  CDate date=CDate(1,5,year);
  date=date+(6-date.getWeekday()+7);
/*
** Fällt Muttertag auf Pfingsten?
** => Muttertag eine Woche vorverlegen
*/
  if(date==calcWhitsun(year))
    date=date-7;
  return(date);
}
```

Listing 282: Die Methode calcMothersDay

Sie finden die Methoden der Klasse `CDate` auf der CD in den `CDate`-Dateien.

54 Wie viele Kalenderwochen besitzt ein Jahr?

Anmerkung 2 zu Rezept 1.3.4 der DIN1355 besagt: »Nur diejenigen Kalenderjahre, die mit einem Donnerstag beginnen oder enden, haben 53 Kalenderwochen.«

Alle anderen Jahre besitzen damit 52 Kalenderwochen. Wir müssen also nur prüfen, ob der 1.1. und/oder der 31.12. des entsprechenden Jahres auf einen Donnerstag fällt.

calcCalendarWeeksPerYear

Dazu erweitern wir die Klasse `CDate` um die Methode `calcCalendarWeeksPerYear`:

```
int CDate::calcCalendarWeeksPerYear(int year) {
  if((CDate(1,1,year).getWeekday()==3)||
     (CDate(31,12,year).getWeekday()==3))
    return(53);
  else
    return(52);
}
```

Listing 283: Die statische Methode calcCalendarWeeksPerYear

Sie finden `calcCalendarWeeksPerYear` auf der CD in den `CDate`-Dateien.

55 Mit welchem Datum beginnt eine bestimmte Kalenderwoche?

Rezept 1.3.3 der DIN1355 besagt: »Als erste Kalenderwoche eines Jahres zählt diejenige Woche, in die mindestens 4 der ersten 7 Januartage fallen.«

Deswegen muss der 4.1. immer in der ersten Kalenderwoche liegen. Von da aus ist es ein Leichtes, das Datum des Montags dieser Woche zu ermitteln und von da aus auf den Beginn der anderen Kalenderwochen zu schließen.

getBeginOfCalendarWeek

Die Klasse `CDate` wird um die Methode `getBeginOfCalendarWeek` erweitert:

```
CDate CDate::getBeginOfCalendarWeek(int year, int cweek) {

/*
** Wochentag des 4.1. ermitteln
*/
  CDate begin(4,1,year);
  int day=begin.getWeekday();

/*
** Datum des Montags der Woche ermitteln,
** in der der 4.1. liegt
*/
  begin=begin-day;

/*
** Anfang der gewünschten KW ermitteln
*/
  begin=begin+((cweek-1)*7);
  return(begin);
}
```

Listing 284: Die Methode getBeginOfCalendarWeek

Sie finden `getBeginOfCalendarWeek` als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien.

56 In welche Kalenderwoche fällt ein bestimmtes Datum?

Um die KW eines Datums zu bestimmen, ermitteln wir zunächst den Montag der Woche, in der das Datum liegt. Anschließend werden die Tage zwischen diesem Montag und dem Montag der ersten KW des Jahres berechnet.

Da es sich um ganze Wochen handelt, ist diese Differenz problemlos durch 7 teilbar und wir haben die aktuelle Kalenderwoche minus 1.

getCalendarWeek

Die Methode `getCalendarWeek` der Klasse `CDate` erledigt diese Aufgabe:

```
int CDate::getCalendarWeek() const {

/*
** Montag der Woche, in der das Datum liegt, ermitteln
*/
  CDate begin=*this-getWeekday();

/*
** Tagesdifferenz zwischen KW des Datums und
** der ersten KW ermitteln
*/
  int diff=begin-getBeginOfCalendarWeek(m_year,1);

/*
** KW des Datums ermitteln
*/
  return((diff/7)+1);
}
```

Listing 285: Die Methode getCalendarWeek

Sie finden `getCalendarWeek` als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien.

57 Wie kann ein Monat übersichtlich dargestellt werden?

Um eine möglichst vielseitige Darstellung zu erhalten, soll zur Formatierung die Sprache HTML eingesetzt werden. Als Ergebnis soll eine Tabelle wie im folgenden Beispiel erzeugt werden:

Juni 2003							
KW	Mo	Di	Mi	Do	Fr	Sa	So
22	26	27	28	29	30	31	**1**
23	**2**	**3**	**4**	**5**	**6**	**7**	**8**
24	**9**	**10**	**11**	**12**	**13**	**14**	**15**
25	**16**	**17**	**18**	**19**	**20**	**21**	**22**
26	**23**	**24**	**25**	**26**	**27**	**28**	**29**
27	**30**	1	2	3	4	5	6

getMonthSmallAsHtml

Dazu erweitern wir die Klasse `CDate` um die Methode `getMonthSmallAsHtml`, der Jahr und Monat übergeben werden. Der HTML-Code wird in Form eines Strings zurückgeliefert:

```
string CDate::getMonthSmallAsHtml(int year, int month) {

/*
** Farbe der zum Monat gehörenden Tage
*/
  static const string InColor="black";

/*
** Farbe der außerhalb des Monats liegenden Tage
*/
  static const string OutColor="#888888";

  static const string TitleColor="#000066";  // Titelfarbe
  static const int Width=25;                 // Zellenbreite

  string str;
```

Listing 286: Die Methode getMonthSmallAsHtml

```
/*
** Erzeugen der Monatsüberschrift
*/
  str+="<table border=\"1\" cellspacing=\"0\">\n";
  str+="<tr>\n";
  str+="<th colspan=\"8\"><font color=\""+TitleColor+"\">";
  str+=CDate(1,month,1).getMonthName()+" "+toString(year)+"</th>\n";
  str+="</tr>\n";

/*
** Erzeugen der Wochentags-Abkürzungen
*/
  str+="<tr>\n";
  str+="<th width=\""+toString(Width)+"\">KW</th>\n";
  str+="<th width=\""+toString(Width)+"\">Mo</th>\n";
  str+="<th width=\""+toString(Width)+"\">Di</th>\n";
  str+="<th width=\""+toString(Width)+"\">Mi</th>\n";
  str+="<th width=\""+toString(Width)+"\">Do</th>\n";
  str+="<th width=\""+toString(Width)+"\">Fr</th>\n";
  str+="<th width=\""+toString(Width)+"\">Sa</th>\n";
  str+="<th width=\""+toString(Width)+"\">So</th>\n";
  str+="</tr>\n";


  CDate date(1,month,year);
  date=date-date.getWeekday();
  int calweek=date.getCalendarWeek();

/*
** Zeilenschleife
*/
  do {

/*
** Ausgabe der KW
*/
    str+="<tr>\n";
    str+="<th align=\"right\">"+toString(calweek)+"</th>\n";

/*
** Spaltenschleife => Ausgabe der Tageszelle
```

Listing 286: Die Methode getMonthSmallAsHtml (Forts.)

```
*/
    for(int i=0; i<7; i++,date=date+1) {
      str+="<td align=\"right\">";
      if(date.getMonth()==month)
        str+="<font color=\""+InColor+"\">";
      else
        str+="<font color=\""+OutColor+"\">";
      str+=toString(date.getDay());
      str+="</font></td>\n";
    }
    str+="</tr>\n";
    calweek++;
  } while(date.getMonth()==month);

  str+="</table>\n";
  return(str);
}
```

Listing 286: Die Methode getMonthSmallAsHtml (Forts.)

Die Methode `getMonthSmallAsHtml` macht von `getMonthName` Gebrauch, die im Folgenden aufgeführt ist.

```
string CDate::getMonthName() const{
  switch(m_month) {
    case 1: return("Januar");
    case 2: return("Februar");
    case 3: return("Maerz");
    case 4: return("April");
    case 5: return("Mai");
    case 6: return("Juni");
    case 7: return("Juli");
    case 8: return("August");
    case 9: return("September");
    case 10: return("Oktober");
    case 11: return("November");
    case 12: return("Dezember");
    default: throw out_of_range("getWeekdayName");
  }
}
```

Listing 287: Die Methode getMonthName

saveStringAsHtmlFile

Um aus dem HTML-formatierten String ein komplettes HTML-Dokument zu erzeugen, implementieren wir eine Funktion `saveStringAsHtmlFile`, die einen String als HTML-Dokument in eine Datei speichert:

```
bool saveStringAsHtmlFile(const std::string &str,
                          const std::string &filename) {

  ofstream os(filename.c_str());
  if(!os.is_open())
    return(false);

  os << "<html>\n<head>\n";
  os << "<title>Generated by saveStringAsHtml()</title>\n";
  os << "</head>\n\n";
  os << "<body>\n";
  os << str;
  os << "</body>\n\n</html>";

  os.close();
  return(true);
}
```

Listing 288: Die Funktion saveStringAsHtmlFile

Sie finden die Methoden der Klasse `CDate` auf der CD in den `CDate`-Dateien.

Das folgende Beispiel fragt nach Monat und Jahr und erstellt damit eine Monatsübersicht als HTML-Datei:

```
#include "CDate.h"
#include "StringFunctions.h"
#include "HTMLFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  int month,year;
  cout << "Monat:";
```

Listing 289: Ein Beispiel zu getMonthSmallAsHtml

```
  cin >> month;
  cout << "Jahr :";
  cin >> year;
  string filename="Month"+toString(month,2,'0')+"-"+
                         toString(year,4,'0')+".html";
  string smonth=CDate::getMonthSmallAsHtml(year, month);
  saveStringAsHtmlFile(smonth, filename);
  cout << "Dokument " << filename << " wurde erzeugt." << endl;
}
```

Listing 289: Ein Beispiel zu getMonthSmallAsHtml (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0414.cpp*.

58 Wie kann ein Monat detailliert dargestellt werden?

Wir verfolgen denselben Ansatz wie im vorherigen Rezept, nur dass eine Darstellung wie in Abbildung 30 erzeugt werden soll.

getMonthDetailedAsHtml

Wir implementieren dazu die Methode `getMonthDetailedAsHtml`:

```
string CDate::getMonthDetailedAsHtml(int year, int month) {

/*
** Hintergrundfarbe der zum Monat gehörenden Tage
*/
  static const string InBg="white";

/*
** Hintergrundfarbe der außerhalb des Monats liegenden Tage
*/
  static const string OutBg="gray";

  static const string TitleColor="#000066";  // Titelfarbe
  static const int Size=100;                 // Zellenbreite

  string str;

/*
** Erzeugen der Monatsüberschrift
```

Listing 290: Die Methode getMonthDetailedAsHtml

```
*/
  str+="<table border=\"1\" cellspacing=\"0\">\n";
  str+="<tr>\n";
  str+="<th colspan=\"8\"><font color=\""+TitleColor+"\">";
  str+=CDate(1,month,1).getMonthName()+" "+toString(year)+"</th>\n";
  str+="</tr>\n";

/*
** Erzeugen der Wochentag-Texte
*/
  str+="<tr>\n";
  str+="<th>KW</th>\n";
  str+="<th width=\""+toString(Size)+"\">Montag</th>\n";
  str+="<th width=\""+toString(Size)+"\">Dienstag</th>\n";
  str+="<th width=\""+toString(Size)+"\">Mittwoch</th>\n";
  str+="<th width=\""+toString(Size)+"\">Donnerstag</th>\n";
  str+="<th width=\""+toString(Size)+"\">Freitag</th>\n";
  str+="<th width=\""+toString(Size)+"\">Samstag</th>\n";
  str+="<th width=\""+toString(Size)+"\">Sonntag</th>\n";
  str+="</tr>\n";


  CDate date(1,month,year);
  date=date-date.getWeekday();
  int calweek=date.getCalendarWeek();

/*
** Zeilenschleife
*/
  do {

/*
** Ausgabe der KW
*/
    str+="<tr>\n";
    str+="<th align=\"right\" height=\""+
         toString(Size)+"\">"+toString(calweek)+"</th>\n";

/*
** Spaltenschleife => Ausgabe der Tageszelle
*/
    for(int i=0; i<7; i++,date=date+1) {
```

Listing 290: Die Methode getMonthDetailedAsHtml (Forts.)

```
      str+="<td align=\"right\" valign=\"top\" ";
      if(date.getMonth()==month)
        str+="bgcolor=\""+InBg+"\">";
      else
        str+="bgcolor=\""+OutBg+"\">";
      str+=toString(date.getDay());
      str+="</td>\n";
    }
    str+="</tr>\n";
    calweek++;
  } while(date.getMonth()==month);

  str+="</table>\n";
  return(str);
}
```

Listing 290: Die Methode getMonthDetailedAsHtml (Forts.)

Juni 2003							
KW	**Montag**	**Dienstag**	**Mittwoch**	**Donnerstag**	**Freitag**	**Samstag**	**Sonntag**
22	26	27	28	29	30	31	1
23	2	3	4	5	6	7	8
24	9	10	11	12	13	14	15
25	16	17	18	19	20	21	22
26	23	24	25	26	27	28	29
27	30	1	2	3	4	5	6

Abbildung 30: Detaillierte Übersicht eines Monats

Sie finden `getMonthDetailedAsHtml` als statische Methode der Klasse `CDate` auf der CD in den `CDate`-Dateien. Die Methode `getMonthDetailedAsHtml` macht von `getMonthName` aus Rezept 57 Gebrauch.

59 Wie kann eine Uhrzeit vernünftig repräsentiert werden?

Wir haben in Rezept 48 bereits die `tm`-Struktur kennen gelernt. Sie deckt zwar auch Uhrzeit ab, aber ihre Genauigkeit ist auf Sekunden begrenzt.

Wir wollen daher eine Klasse `CTime` implementieren, die eine Genauigkeit im Millisekundenbereich zulässt. Wir verwenden dazu folgende Attribute:

- `m_hour` Stunden (0-23)
- `m_minute` Minuten (0-59
- `m_second` Sekunden (0-59)
- `m_millisecond` Millisekunden (0-999)
- `m_millisecondsPast` Millisekunden seit 0:00:00:0000 (0-86399999)

Die Klasse wird mit inline-Zugriffsmethoden für diese Attribute ausgestattet.

Klassendefinition

```
class CTime {
private:
  int m_hour;
  int m_minute;
  int m_second;
  int m_millisecond;
  long m_millisecondsPast;

  void calcMillisecondsPast();
  void calcTime();
  void checkRanges();

public:

  CTime(int hour, int minute, int second=0, int millisecond=0);
```

Listing 291: Die Klassendefinition von CTime

```
CTime(long);
CTime(const std::string &time);
CTime();

int getHour() const {
  return(m_hour);
}

int getMinute() const {
  return(m_minute);
}

int getSecond() const {
  return(m_second);
}

int getMillisecond() const {
  return(m_millisecond);
}

void setHour(long w) {
  m_hour=w;
  checkRanges();
  calcMillisecondsPast();
}

void setMinute(long w) {
  m_minute=w;
  checkRanges();
  calcMillisecondsPast();
}

void setSecond(long w) {
  m_second=w;
  checkRanges();
  calcMillisecondsPast();
}

void setMillisecond(long w) {
  m_millisecondsecond=w;
  checkRanges();
```

Listing 291: Die Klassendefinition von CTime (Forts.)

```
    calcMillisecondsPast();
  }

  void setMillisecondsPast(long w) {
    m_millisecond=w;
    calcTime();
  }

  long getMillisecondsPast() const {
    return(m_millisecondsPast);
  }

  std::string getHHMM() const;
  std::string getHHMMSS() const;
  std::string getHHMMSSMMM() const;
  CTime operator+(long mseconds) const;
  CTime operator-(long mseconds) const;
  CTime operator+(const CTime &z) const;
  CTime operator-(const CTime &z) const;
  bool operator<(const CTime &z) const;
  bool operator==(const CTime &z) const;
  bool isEqual(const CTime &z, long eps) const;
};
```

Listing 291: Die Klassendefinition von CTime (Forts.)

Konstruktoren

Die Klasse besitzt einen Konstruktor, um ein Objekt aus Stunden, Minuten, Sekunden und Millisekunden zu erzeugen. Dabei sind die Parameter für Sekunden und Millisekunden mit den Standard-Werten 0 versehen:

```
CTime::CTime(int hour, int minute, int second, int millisecond)
:m_hour(hour), m_minute(minute),
 m_second(second), m_millisecond(millisecond) {

  checkRanges();
  calcMillisecondsPast();
}
```

Listing 292: Konstruktor für Stunden, Minuten, Sekunden und Millisekunden

Ein weiterer Konstruktor berechnet aus Millisekunden die Uhrzeit. Dabei wird der Wertebereich als Ring aufgefasst:

```
CTime::CTime(long milliseconds)
: m_millisecondsPast(milliseconds) {
  if(m_millisecondsPast<0) {
    m_millisecondsPast+=24*60*60*1000;
  } else if(m_millisecondsPast>=24*60*60*1000) {
    m_millisecondsPast%=24*60*60*1000;
  }

  calcTime();
}
```

Listing 293: Ein Konstruktor für Millisekunden

Der nächste Konstruktor erlaubt die Angabe der Uhrzeit als String. Dabei darf die Uhrzeit dreiteilig ohne Millisekunden oder vierteilig mit Millisekunden sein.

```
CTime::CTime(const string &time) {

/*
** Uhrzeit in Einzelteile zerlegen
*/
  vector<string> components=chopIntoWords(time," :");

/*
** Es dürfen nur drei oder vier Einzelteile sein.
*/
  if((components.size()<3)||(components.size()>4))
    throw invalid_argument("CTime::CTime(string)");

/*
** Einzelteile umwandeln
*/
  m_hour=atol(components[0].c_str());
  m_minute=atol(components[1].c_str());
  m_second=atol(components[2].c_str());

/*
```

Listing 294: Ein Konstruktor für Uhrzeiten im String-Format

```
** Gegebenenfalls noch die Millisekunden umwandeln
*/
  if(components.size()==4)
    m_millisecond=atol(components[3].c_str());
  else
    m_millisecond=0;

  checkRanges();
  calcMillisecondsPast();
}
```

Listing 294: Ein Konstruktor für Uhrzeiten im String-Format (Forts.)

Der Standard-Konstruktor schließlich initialisiert das Objekt mit der aktuellen Systemzeit:

```
CTime::CTime() {
  time_t t;
  time(&t);
  tm *ts;
  ts=localtime(&t);
  m_hour=ts->tm_hour;
  m_minute=ts->tm_min;
  m_second=ts->tm_sec;
  m_millisecond=0;

  checkRanges();
  calcMillisecondsPast();
}
```

Listing 295: Der Standard-Konstruktor

checkRanges

Die Methode `checkRanges` überprüft die einzelnen Attribute auf gültige Werte. Sollte ein Wert außerhalb des gültigen Bereichs liegen, wird die STL-Ausnahme `out_of_range` geworfen.

```
void CTime::checkRanges() {
  if((m_hour<0)||(m_hour>23))
```

Listing 296: Die Methode checkRanges

```
    throw out_of_range("CTime::checkRanges");
  if((m_minute<0)||(m_minute>59))
    throw out_of_range("CTime::checkRanges");
  if((m_second<0)||(m_second>59))
    throw out_of_range("CTime::checkRanges");
  if((m_millisecond<0)||(m_millisecond>999))
    throw out_of_range("CTime::checkRanges");
}
```

Listing 296: Die Methode checkRanges (Forts.)

calcMillisecondsPast

Die Methode `calcMillisecondsPast` berechnet aus den Stunden, Minuten, Sekunden und Millisekunden die seit 0:00 vergangenen Millisekunden:

```
void CTime::calcMillisecondsPast() {
  m_millisecondsPast=
    (((((m_hour*60)+m_minute)*60)+m_second)*1000)+
    m_millisecond;
}
```

Listing 297: Die Methode calcMillisecondsPast

calcTime

Das Gegenstück zu `calcMillisecondsPast` bildet die Methode `calcTime`, die aus den seit 0:00 vergangenen Millisekunden die Stunden, Minuten, Sekunden und Millisekunden berechnet:

```
void CTime::calcTime() {
  long mseconds=m_millisecondsPast;

  m_millisecond=mseconds%1000;
  mseconds/=1000;
  m_second=mseconds%60;
  mseconds/=60;
  m_minute=mseconds%60;
  m_hour=mseconds/60;
```

Listing 298: Die Methode calcTime

```
  checkRanges();
}
```

Listing 298: Die Methode calcTime (Forts.)

getHHMM

Um die Uhrzeit ausgeben zu können existieren drei Methoden, die die Uhrzeit im gewünschten Format als String zurückliefern:

```
string CTime::getHHMM() const {

  stringstream str;

  str.fill('0');
  str << setw(2) << m_hour << ":";
  str << setw(2) << m_minute;
  return(str.str());
}
```

Listing 299: Die Methode getHHMM

getHHMMSS

```
string CTime::getHHMMSS() const {

  stringstream str;

  str.fill('0');
  str << setw(2) << m_hour << ":";
  str << setw(2) << m_minute << ":";
  str << setw(2) << m_second;
  return(str.str());
}
```

Listing 300: Die Methode getHHMMSS

getHHMMSSMMM

```
string CTime::getHHMMSSMMM() const {

  stringstream str;

  str.fill('0');
  str << setw(2) << m_hour << ":";
  str << setw(2) << m_minute << ":";
  str << setw(2) << m_second << ":";
  str << setw(3) << m_millisecond;
  return(str.str());
}
```

Listing 301: Die Methode getHHMMSSMMM

Rechenoperatoren

Zusätzlich werden Rechenoperatoren implementiert, die es erlauben, Summen und Differenzen von Uhrzeiten und von Uhrzeiten und Millisekunden zu bilden.

```
CTime CTime::operator+(long mseconds) const {
  return(CTime(m_millisecondsPast+mseconds));
}

//*********

CTime CTime::operator+(const CTime &z) const {
  return(CTime(m_millisecondsPast+m_millisecondsPast));
}

//*********

CTime CTime::operator-(long mseconds) const {
  return(CTime(m_millisecondsPast-mseconds));
}

//*********

CTime CTime::operator-(const CTime &z) const {
```

Listing 302: Die Rechenoperatoren von CTime

```
  return(CTime(labs(m_millisecondsPast-m_millisecondsPast)));
}
```

Listing 302: Die Rechenoperatoren von CTime (Forts.)

Vergleichsoperatoren

Um Uhrzeiten vom Typ `CTime` vergleichen zu können, werden die Vergleichsoperatoren < und == implementiert, alle anderen sind durch die Templates aus `rel_ops` abgedeckt:

```
bool CTime::operator<(const CTime &z) const {
  return(m_millisecondsPast<z.m_millisecondsPast);
}

//*********

bool CTime::operator==(const CTime &z) const {
  return(m_millisecondsPast==z.m_millisecondsPast);
}
```

Listing 303: Die Vergleichsoperatoren von CTime

isEqual

Das Prüfen auf Gleichheit birgt insofern eine Problematik, weil zwei Uhrzeiten in den seltensten Fällen bis auf die Millisekunde gleich sind. Deswegen implementieren wir eine Methode `isEqual`, bei der die Möglichkeit besteht, eine Grauzone zu definieren. Sollten zwei Uhrzeiten weniger als `eps` Millisekunden auseinander liegen, dann erachtet `isEqual` diese als gleich:

```
bool CTime::isEqual(const CTime &z, long eps) const {
  return(labs(m_millisecondsPast-z.m_millisecondsPast)<=eps);
}
```

Listing 304: Die Methode isEqual

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`.

Ein Objekt der Klasse `CTime` ist komplett durch den Wert `m_millisecondsPast` dargestellt. Alle anderen Attribute lassen sich daraus errechnen. Es muss jedoch darauf geachtet werden, dass diese Berechnung beim Laden durchgeführt wird.

Die Stream-Operatoren müssen als Freunde der Klasse deklariert werden.

```
CBinaryOStream &operator<<(CBinaryOStream &os, const CTime &t) {
  os << t.m_millisecondsPast;
  return(os);
}

//*********

CBinaryIStream &operator>>(CBinaryIStream &is, CTime &t) {
  is >> t.m_millisecondsPast;
  t.calcTime();
  t.checkRanges();
  return(is);
}
```

Sie finden die Klasse `CTime` auf der CD in den `CTime`-Dateien.

60 Wie kann eine Kalenderwoche dargestellt werden?

Wir wollen eine wie in Abbildung 31 dargestellte Formatierung erzeugen.

getDDMM

Dazu ergänzen wir die Klasse `CDate` zunächst um die Ausgabe-Methode `getDDMM`, um das in der Abbildung vorgegebene Datumsformat erzeugen zu können:

```
string CDate::getDDMM() const {

  stringstream str;

  str.fill('0');
  str << setw(2) << m_day << ".";
  str << setw(2) << m_month << ".";
```

Listing 305: Die Methode getDDMM

```
    return(str.str());
}
```

Listing 305: Die Methode getDDMM (Forts.)

getCalendarWeekAsHtml

Die Methode getCalendarWeekAsHtml besitzt folgende Parameter:

- `year` das Jahr, in dem die Kalenderwoche liegt
- `cweek` die darzustellende Kalenderwoche
- `begin` Uhrzeit, von der ab dargestellt werden soll (Standard: 18:00)
- `end` Uhrzeit, bis zu der dargestellt werden soll (Standard: 8:00)
- `intervall` Größe der Zeitintervalle (Standard: 30)
- `width` Breite der Zellen (Standard: 100)
- `height` Höhe der Zellen (Standard: 25)

2003 KW 23	Mo, 02.06.	Di, 03.06.	Mi, 04.06.	Do, 05.06.	Fr, 06.06.	Sa, 07.06.	So, 08.06.
08:00							
08:30							
09:00							
09:30							
10:00							
10:30							
11:00							
11:30							
12:00							
12:30							
13:00							
13:30							
14:00							
14:30							
15:00							
15:30							
16:00							
16:30							
17:00							
17:30							

Abbildung 31: Darstellung einer Kalenderwoche

Und hier kommt die Methode:

```
string CDate::getCalendarWeekAsHtml(int year, int cweek,
                                    const CTime &begin,
                                    const CTime &end,
                                    int intervall,
                                    int width,
                                    int height) {

  string str;

  str+="<table border=\"1\" cellspacing=\"0\">\n";

/*
** Erzeugen der Wochentag-Texte
*/
  str+="<tr>\n";
  str+="<th>"+toString(year)+"<br />KW "+toString(cweek)+"</th>\n";
  CDate date=getBeginOfCalendarWeek(year, cweek);
  str+="<th width=\""+toString(width)+
       "\">Mo, "+date.getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">Di, "+(date+1).getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">Mi, "+(date+2).getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">Do, "+(date+3).getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">Fr, "+(date+4).getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">Sa, "+(date+5).getDDMM()+"</th>\n";
  str+="<th width=\""+toString(width)+
       "\">So, "+(date+6).getDDMM()+"</th>\n";

  CTime time=begin;

/*
** Zeilenschleife
*/
  do {
```

Listing 306: getCalendarWeekAsHtml

```
/*
** Ausgabe der Uhrzeit
*/
    str+="<tr>\n";
    str+="<th align=\"right\" height=\""+
         toString(height)+"\">"+time.getHHMM()+"</th>\n";

/*
** Spaltenschleife
*/
    for(int i=0; i<7; i++,date=date+1) {
      str+="<td> </td>\n";
    }
    str+="</tr>\n";
    time=time+intervall*60*1000;
  } while(time<end);

  str+="</table>\n";
  return(str);
}
```

Listing 306: getCalendarWeekAsHtml (Forts.)

Das folgende Beispiel fragt nach dem Jahr und der Kalenderwoche und erstellt daraus ein HTML-Dokument:

```
#include "CDate.h"
#include "StringFunctions.h"
#include "HTMLFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  int cw,year;
  cout << "KWoche:";
  cin >> cw;
  cout << "Jahr  :";
  cin >> year;
```

Listing 307: Ein Beispiel zu getCalendarWeekAsHtml

```
  string filename="CW"+toString(cw,2,'0')+"-"+
                      toString(year,4,'0')+".html";
  string smonth=CDate::getCalendarWeekAsHtml(year, cw);
  saveStringAsHtmlFile(smonth, filename);
  cout << "Dokument " << filename << " wurde erzeugt." << endl;
}
```

Listing 307: Ein Beispiel zu getCalendarWeekAsHtml (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0417.cpp*.

61 Wie kann ein Zeitpunkt vernünftig repräsentiert werden?

Da wir bereits die Klasse `CDate` zur Repräsentation eines beliebigen Datums und die Klasse `CTime` zur Repräsentation einer beliebigen Uhrzeit an einem Tag besitzen, brauchen wir sie nur zu einer Klasse zusammenzufassen.

CMoment

Wir erstellen eine Klasse `CMoment`, die diese Leistung erbringt. Zunächst die Klassendefinition mit entsprechenden, als inline deklarierten Zugriffsmethoden:

Klassendefinition

```
class CMoment {
private:
  CDate m_date;
  CTime m_time;

public:

/*
** Konstruktoren
*/
  CMoment();
  CMoment(const std::string &moment);
  CMoment(const std::string &date, const std::string &time);
  CMoment(const CTimevector &z);
  CMoment(const CDate &d);
```

Listing 308: Die Klassendefinition von CMoment

```
  CMoment(const CDate &d, const CTime &z);
  CMoment(int year,
          int month,
          int day,
          int hour=0,
          int minute=0,
          int second=0,
          int msecond=0);

/*
** Zeit und Datum per Methode
*/
  CDate getDate() const {
    return(m_date);
  }

  CTime getTime() const {
    return(m_time);
  }

  void setDate(const CDate &d) {
    m_date=d;
  }

  void setTime(const CTime &z) {
    m_time=z;
  }

/*
** Lesen und Setzen der einzelnen Attribute
*/
  int getHour() const {
    return(m_time.getHour());
  }

  int getMinute() const {
    return(m_time.getMinute());
  }

  int getSecond() const {
    return(m_time.getSecond());
```

Listing 308: Die Klassendefinition von CMoment (Forts.)

```
  }

  int getMillisecond() const {
    return(m_time.getMillisecond());
  }

  int getMillisecondsPast() const {
    return(m_time.getMillisecondsPast());
  }

  void setHour(long w) {
    m_time.setHour(w);
  }

  void setMinute(long w) {
    m_time.setMinute(w);
  }

  void setSecond(long w) {
    m_time.setSecond(w);
  }

  void setMillisecond(long w) {
    m_time.setMillisecond(w);
  }

  int getYear() const {
    return(m_date.getYear());
  }

  int getMonth() const {
    return(m_date.getMonth());
  }

  int getDay() const {
    return(m_date.getDay());
  }

  int getDaysPast() const {
    return(m_date.getDaysPast());
  }
```

Listing 308: Die Klassendefinition von CMoment (Forts.)

```
  void setYear(long w) {
    m_date.setYear(w);
  }

  void setMonth(long w) {
    m_date.setMonth(w);
  }

  void setDay(long w) {
    m_date.setDay(w);
  }


  std::string getDDMMYYYYHHMMSS() const;

  CMoment operator+(const CTimevector &z) const;
  CMoment operator-(const CTimevector &z) const;
  bool operator<(const CMoment &z) const;
  bool operator==(const CMoment &z) const;
  bool isEqual(const CMoment &z, long eps) const;
};
```

Listing 308: Die Klassendefinition von CMoment (Forts.)

Konstruktoren

Die Konstruktoren sind relativ leicht zu erstellen, weil die tatsächliche Arbeit von den Konstruktoren der Klassen `CDate` und `CTime` erledigt wird.

Der Standard-Konstruktor von `CMoment` initialisiert das Objekt mit dem aktuellen Datum und der aktuellen Uhrzeit.

```
CMoment::CMoment()
  : m_date(), m_time() {
}

//*********

CMoment::CMoment(const CDate &d)
  : m_date(d), m_time(0,0,0,0) {
```

Listing 309: Die Konstruktoren von CMoment

```
}

//*********

CMoment::CMoment(const CDate &d, const CTime &z)
  : m_date(d), m_time(z) {
}

//*********

CMoment::CMoment(int year,
                 int month,
                 int day,
                 int hour,
                 int minute,
                 int second,
                 int msecond)
: m_date(day, month, year),
  m_time(hour, minute, second, msecond) {
}

//*********

CMoment::CMoment(const CTimevector &z)
: m_date(z.getDays()), m_time(z.getMilliseconds()) {
}

//*********

CMoment::CMoment(const std::string &date, const std::string &time)
: m_date(date), m_time(time) {
}

//*********

CMoment::CMoment(const std::string &moment) {
  vector<string> c=chopIntoWords(moment, " .:-");
  if(c.size()!=8)
    throw invalid_argument("CMoment::CMoment(string)");
  m_date=CDate(c[1]+" "+c[2]+" "+c[3]);
```

Listing 309: Die Konstruktoren von CMoment (Forts.)

```
  m_time=CTime(c[4]+":"+c[5]+":"+c[6]);
}
```

Listing 309: Die Konstruktoren von CMoment (Forts.)

Der letzte Konstruktor initialisiert das Objekt mit einem `CTimevector`-Objekt. Diese Klasse werden wir später noch im Detail besprechen.

getDDMMYYYYHHMMSS

Zur Ausgabe eines `CMoment`-Objekts implementieren wir die Methode `getDDMMYYYY-HHMMSS`, die auf die Ausgabe-Methoden von `CDate` und `CTime` zurückgreift und das Ergebnis als String zurückliefert.

```
string CMoment::getDDMMYYYYHHMMSS() const {
  return(m_date.getDDMMYYYY()+" "+m_time.getHHMMSS());
}
```

Listing 310: Die Methode getDDMMYYYYHHMMSS

Andere Ausgaben von `CMoment`-Objekten können selbst zusammengestellt werden, indem die Ausgabe-Methoden von `CDate` und `CTime` direkt angesprochen werden.

getRFC1123Date

Wir wollen auch das im Internet übliche, in RFC1123 definierte Datums-Format unterstützen, das Datum und Uhrzeit in einem String mit fester Länge ausgibt. Hier ein Beispiel:

```
Fri, 30 May 2003 16:50:11 GMT
```

Üblicherweise wird das Datum in GMT ausgegeben. Die Klasse `CMoment` ist mit entsprechenden Methoden ausgestattet, um MEZ in GTM und zurück umzuwandeln.

```
string CMoment::getRFC1123Date() const {
  static const char* const months[]={"Jan","Feb", "Mar",
                                     "Apr", "May", "Jun",
                                     "Jul", "Aug", "Sep",
```

Listing 311: Die Methode getRFC1123Date

```
                                    "Oct", "Nov", "Dec"};

  static const char* const days[]={"Mon","Tue", "Wed", "Thu",
                                   "Fri", "Sat", "Sun"};

  stringstream str;
  str.fill('0');

  str << days[m_date.getWeekday()] << ", ";
  str << setw(2) << m_date.getDay() << " ";
  str << months[m_date.getMonth()-1] << " ";
  str << setw(4) << m_date.getYear() << " ";
  str << setw(2) << m_time.getHour() << ":";
  str << setw(2) << m_time.getMinute() << ":";
  str << setw(2) << m_time.getSecond() << " GMT";
  return(str.str());
}
```

Listing 311: Die Methode getRFC1123Date (Forts.)

Rechenoperatoren

Die Rechenoperatoren reduzieren die Problematik auf die `CTimevector`-Rechenoperatoren. Genauere Details besprechen wir im Zusammenhang mit der `CTimevector`-Klasse.

```
CMoment CMoment::operator+(const CTimevector &z) const {
  return(CTimevector(*this)+z);
}

//*********

CMoment CMoment::operator-(const CTimevector &z) const {
  return(CTimevector(*this)-z);
}
```

Listing 312: Die Rechenoperatoren von CMoment

Vergleichsoperatoren

Auch die Vergleichsoperatoren greifen auf die Klasse `CTimevector` zurück:

```
bool CMoment::operator<(const CMoment &z) const {
  return(CTimevector(*this)<z);
}

//*********

bool CMoment::operator==(const CMoment &z) const {
  return(CTimevector(*this)==z);
}
```

Listing 313: Die Vergleichsoperatoren von CMoment

isEqual

Die Schwierigkeit, dass Uhrzeiten selten bis auf die Millisekunde identisch sind, tritt auch bei `CMoment` auf, deswegen implementieren wir hier ebenfalls eine `isEqual`-Methode.

```
bool CMoment::isEqual(const CMoment &z, long eps) const {
  return((m_date==z.m_date)&&(m_time.isEqual(z.m_time,eps)));
}
```

Listing 314: Die Methode isEqual

GMT-Umwandlung

Häufig (besonders im Internet) finden sich Zeitangaben in GMT (Abk. für Greenwich Mean Time). Aus dieser Zeit muss die für Deutschland geltende Zeit noch bestimmt werden. Je nachdem, ob wir gerade Sommerzeit haben oder nicht, muss auf GMT 2 oder 1 Stunde aufaddiert werden, um MESZ bzw. MEZ zu bekommen.

Die Methoden `convertGmtToMet` und `convertMetToGmt` wandeln die Uhrzeiten entsprechend um.

```
CMoment CMoment::convertGmtToMet() const {
  time_t t;
  time(&t);
  tm *ts;
  ts=localtime(&t);
```

Listing 315: Die Methoden zur Umwandlung von GMT und MEZ

```
/*
** Wenn Sommerzeit, dann zwei Stunden,
** andernfalls eine Stunde addieren
*/
  if(ts->tm_isdst)
    return(*this+CTimevector(0,7200000));
  else
    return(*this+CTimevector(0,3600000));
}

//*********

CMoment CMoment::convertMetToGmt() const {
  time_t t;
  time(&t);
  tm *ts;
  ts=localtime(&t);

/*
** Wenn Sommerzeit, dann zwei Stunden,
** andernfalls eine Stunde abziehen
*/
  if(ts->tm_isdst)
    return(*this-CTimevector(0,7200000));
  else
    return(*this-CTimevector(0,3600000));
}
```

Listing 315: Die Methoden zur Umwandlung von GMT und MEZ (Forts.)

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`.

Da für `CMoment`-Komponenten `CTime` und `CDate` bereits Stream-Operatoren existieren, gestaltet sich die Implementierung recht simpel.

Die Stream-Operatoren müssen als Freunde der Klasse deklariert werden.

```
CBinaryOStream &operator<<(CBinaryOStream &os, const CMoment &m) {
  os << m.m_date;
  os << m.m_time;
  return(os);
}

CBinaryIStream &operator>>(CBinaryIStream &is, CMoment &m) {
  is >> m.m_date;
  is >> m.m_time;
  return(is);
}
```

Listing 316: Die Stream-Operatoren von CMoment

Sie finden die Klasse `CMoment` auf der CD in den `CMoment`-Dateien.

CTimevector

Wenn wir mit der Klasse `CMoment` arbeiten, kann es durchaus vorkommen, dass wir auf einen Zeitpunkt einige Minuten oder Stunden hinzuaddieren möchten. Angenommen, ein `CMoment`-Objekt soll zwei Stunden vorgesetzt werden, dann müssten wir 0 Tage und 7200000 Millisekunden hinzuaddieren. In unserer Schreibweise wären das 00.00.0000 2:00:00:000. Der kleinstmögliche Wert eines `CMoment`-Objekts ist aber 01.01.0001 0:00:00:000, also viel zu groß.

Die Klasse `CMoment` eignet sich daher nicht für die Darstellung solcher Zeitspannen. Wir benötigen eine Klasse, die mit einem Ortsvektor aus der Mathematik zu vergleichen ist. Und diese Aufgabe übernimmt `CTimevector`. Objekte dieser Klasse besitzen lediglich zwei Attribute: einen Offset für die Tage und einen für die Millisekunden.

Klassendefinition

```
class CTimevector {
private:
  long m_days;
  long m_mseconds;

public:
  CTimevector();
  CTimevector(const CDate &d);
```

Listing 317: Die Klassendefinition von CTimevector

```
  CTimevector(const CTime &z);
  CTimevector(const CMoment &z);
  CTimevector(const CDate &d, const CTime &z);
  CTimevector(long d, long z);

  long getDays() const {
    return(m_days);
  }

  long getMilliseconds() const {
    return(m_mseconds);
  }

  CTimevector operator+(const CTimevector &z) const;
  CTimevector operator-(const CTimevector &z) const;

  bool operator<(const CTimevector &z) const;
  bool operator==(const CTimevector &z) const;
};
```

Listing 317: Die Klassendefinition von CTimevector (Forts.)

Konstruktoren

`CTimevector` besitzt viele Konstruktoren, um Objekte aus allen für uns relevanten Datentypen erzeugen zu können.

```
CTimevector::CTimevector()
: m_days(0), m_mseconds(0)
{}

//*********

CTimevector::CTimevector(const CDate &d)
: m_days(d.getDaysPast()), m_mseconds(0)
{}

//*********

CTimevector::CTimevector(const CTime &z)
: m_days(0), m_mseconds(z.getMillisecondsPast())
```

Listing 318: Die Konstruktoren von CTimevector

```
{}

//*********

CTimevector::CTimevector(const CMoment &z)
: m_days(z.getDaysPast()),
  m_mseconds(z.getMillisecondsPast())
{}

//*********

CTimevector::CTimevector(const CDate &d, const CTime &z)
: m_days(d.getDaysPast()),
  m_mseconds(z.getMillisecondsPast())
{}

//*********

CTimevector::CTimevector(long d, long z)
: m_days(d), m_mseconds(z)
{}
```

Listing 318: Die Konstruktoren von CTimevector (Forts.)

Rechenoperatoren

Die Rechenoperatoren sind etwas aufwändiger, denn sie müssen dafür Sorge tragen, dass bei einem Überlauf der Zeit (23:59:59:999 wird überschritten) der Tag um eins erhöht und die Zeit wieder entsprechend zurückgesetzt wird. Ähnliche Vorkehrungen müssen bei einem Unterlauf getroffen werden.

```
CTimevector CTimevector::operator+(const CTimevector &z) const{
  CTimevector tmp(m_days+z.m_days, m_mseconds+z.m_mseconds);
  if(tmp.m_mseconds>=24*60*60*1000) {
    tmp.m_mseconds%=24*60*60*1000;
    tmp.m_days++;
  }
  return(tmp);
}

//*********
```

Listing 319: Die Rechenoperatoren von CTimevector

```
CTimevector CTimevector::operator-(const CTimevector &z) const{
  CTimevector tmp;

  if(z<*this) {
    tmp.m_days=m_days-z.m_days;
    tmp.m_mseconds=m_mseconds-z.m_mseconds;
    if(tmp.m_mseconds<0) {
      tmp.m_mseconds+=24*60*60*1000;
      tmp.m_days--;
    }
  } else {
    tmp.m_days=z.m_days-m_days;
    tmp.m_mseconds=z.m_mseconds-m_mseconds;
    if(tmp.m_mseconds<0) {
      tmp.m_mseconds+=24*60*60*1000;
      tmp.m_days--;
    }
  }
  return(tmp);
}
```

Listing 319: Die Rechenoperatoren von CTimevector (Forts.)

Vergleichsoperatoren

Die Vergleichsoperatoren müssen bei den Vergleichen beide Komponenten (Tage und Millisekunden) berücksichtigen:

```
bool CTimevector::operator<(const CTimevector &z) const {
  if(m_days<z.m_days)
    return(true);
  if((m_days==z.m_days)&&(m_mseconds<z.m_mseconds))
    return(true);
  return(false);
}

//*********

bool CTimevector::operator==(const CTimevector &z) const {
```

Listing 320: Die Vergleichsoperatoren von CTimevector

```
  return((m_days==z.m_days)&&(m_mseconds==z.m_mseconds));
}
```

Listing 320: Die Vergleichsoperatoren von CTimevector (Forts.)

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

62 Wie kann aus einem Geburtsdatum das Alter bestimmt werden?

Das Alter kann bestimmt werden, indem die Differenz zwischen dem aktuellen Datum und dem Geburtsdatum gebildet wird.

calcAge

Wir erweitern dazu die Klasse `CDate` aus Rezept 48 um die statische Methode `calcAge`:

```
int CDate::calcAge(const CDate &d) {

/*
** Differenz von jetzigem Jahr und
** Geburtsjahr bilden
*/
  CDate today;
  int age=today.getYear()-d.getYear();

/*
** Geburtstag in diesem Jahr noch nicht erreicht?
** => Alter um eins vermindern
*/
  if(today.getMonth()<d.getMonth())
    return(age-1);
  else if((today.getMonth()==d.getMonth())&&(today.getDay()<d.getDay()))
    return(age-1);

  return(age);
}
```

Listing 321: Die statische Methode calcAge von CDate

Für den Fall, dass `calcAge` direkt auf ein `CDate`-Objekt angewendet werden soll, überladen wir die Methode:

```
int CDate::calcAge() {
  return(calcAge(*this));
}
```

Listing 322: Die überladene Variante von calcAge

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

63 Wie kann zu einem Datum das Sternzeichen ermittelt werden?

Die Methode `calcZodiacSign` der Klasse `CDate` bestimmt das Sternzeichen zu einem gegebenen Jahr und Monat und liefert es als String zurück:

```
string CDate::calcZodiacSign(int day, int month) {
  switch(month) {
    case 1:
      if(day<=20)
        return("Steinbock");
      else
        return("Wassermann");

    case 2:
      if(day<=19)
        return("Wassermann");
      else
        return("Fische");

    case 3:
      if(day<=20)
        return("Fische");
      else
        return("Widder");

    case 4:
      if(day<=20)
```

Listing 323: Die statische Methode calcZodiacSign von CDate

```
      return("Widder");
    else
      return("Stier");

  case 5:
    if(day<=21)
      return("Stier");
    else
      return("Zwillinge");

  case 6:
    if(day<=21)
      return("Zwillinge");
    else
      return("Krebs");

  case 7:
    if(day<=22)
      return("Krebs");
    else
      return("Löwe");

  case 8:
    if(day<=23)
      return("Löwe");
    else
      return("Jungfrau");

  case 9:
    if(day<=23)
      return("Jungfrau");
    else
      return("Waage");

  case 10:
    if(day<=23)
      return("Waage");
    else
      return("Skorpion");

  case 11:
```

Listing 323: Die statische Methode calcZodiacSign von CDate (Forts.)

```
      if(day<=22)
        return("Skorpion");
      else
        return("Schütze");

    case 12:
      if(day<=21)
        return("Schütze");
      else
        return("Steinbock");

    default:
      throw(out_of_range("calcZodiacsign"));
  }
}
```

Listing 323: Die statische Methode calcZodiacSign von CDate (Forts.)

Und hier noch die Variante zur Bestimmung des Tierkreiszeichens eines `CDate`-Objekts:

```
string CDate::calcZodiacSign() {
  return(calcZodiacSign(m_day,m_month));
}
```

Listing 324: Die überladene Variante von calcZodiacSign

Sie finden die Klasse `CDate` auf der CD in den `CDate`-Dateien.

64 Wie wird eine URL in ihre Einzelteile zerlegt?

Wir wollen uns dazu auf die Schemata `http:`, `ftp:` und `file:` beschränken.

CUrl

Wir definieren eine Klasse `CUrl`, die gemäß RFC1738 folgende Attribute und Methoden besitzt:

Klassendefinition

```
class CUrl {
public:
  enum Scheme {UNDEFINED=0, PARTIAL, HTTP, FTP, FILE};
  typedef std::vector<std::string> folder_container;
private:
  std::string m_url;

  Scheme m_scheme;
  std::string m_protocol;
  std::string m_username;
  std::string m_password;
  std::string m_domain;
  std::string m_port;
  folder_container m_folders;
  std::string m_parameters;
  std::string m_fragment;
  std::string m_ftptype;
};
  size_t scheme(const insensitive_string &url,
                size_t be);

  size_t user(const insensitive_string &url,
              size_t be);

  size_t password(const insensitive_string &url,
                  size_t be);
```

Listing 325: Die Klassendefinition von CUrl

```
  size_t host(const insensitive_string &url,
              size_t be);

  size_t filehost(const insensitive_string &url,
                  size_t be);

  size_t port(const insensitive_string &url,
              size_t be);

  size_t hostport(const insensitive_string &url,
                  size_t be);

  size_t login(const insensitive_string &url,
               size_t be);

  size_t httppath(const insensitive_string &url,
                  size_t be);

  size_t ftppath(const insensitive_string &url,
                 size_t be);

  size_t filepath(const insensitive_string &url,
                  size_t be);

  size_t search(const insensitive_string &url,
                size_t be);

  size_t fragment(const insensitive_string &url,
                  size_t be);

  size_t ftptype(const insensitive_string &url,
                 size_t be);

  bool divideUrl(Scheme scheme);
  bool isUrlPartial(const std::string &url);

public:
  class EUrlError {};
  CUrl();
```

Listing 325: Die Klassendefinition von CUrl (Forts.)

```
  CUrl(const std::string &url, Scheme s=HTTP);

  void showUrl() const;
};
```

Listing 325: Die Klassendefinition von CUrl (Forts.)

Die Attribute haben folgende Bedeutung:

- `m_scheme` ist vom Typ Scheme und definiert in numerischer Form das Schema der URL.
- `m_protocol` beinhaltet das Protokoll (Schema) als String (z.B. »`http://`«).
- `m_username` beinhaltet bei vorhandenem login den Usernamen.
- `m_password` beinhaltet eventuell das Passwort.
- `m_domain` beinhaltet die komplette Domäne (z.B. `www.addison-wesley.de`).
- `m_port` beinhaltet den Port (z.B. bei Ftp standardmäßig 21).
- `m_folders` ist ein Vektor mit allen Pfad-Teilen der URL.
- `m_parameters` enthält die mit `?` an den Dateinamen gehängten Parameter.
- `m_fragment` nimmt das mit # eingeleitete Fragment auf.
- `m_ftptype` ist der mit `;type=` definierte FTP-Typ.

Zugriffsmethoden

Als Nächstes folgen die Zugriffsmethoden für die Attribute.

```
std::string getUrl() const {
  return(m_url);
}

std::string getProtocol() const {
  return(m_protocol);
}

Scheme getScheme() const {
  return(m_scheme);
}
```

Listing 326: Die Zugriffsmethoden von CUrl

```
std::string getUsername() const {
  return(m_username);
}

std::string getPassword() const {
  return(m_password);
}

std::string getDomain() const {
  return(m_domain);
}

std::string getPort() const {
  return(m_port);
}

std::string getParameters() const {
  return(m_parameters);
}

std::string getFragment() const {
  return(m_fragment);
}

std::string getFtptype() const {
  return(m_ftptype);
}

const std::vector<std::string> &getFolders() const {
  return(m_folders);
}
```

Listing 326: Die Zugriffsmethoden von CUrl (Forts.)

Weil bei einigen Teilen der URL (Schema, Host) die Groß- und Kleinschreibung vernachlässigt werden muss, arbeiten die internen Methoden mit dem `insensitive_string` aus Rezept 17.

Konstruktor

Der Konstruktor besitzt einen optionalen Parameter `s`, mit dem bei einer partiellen URL das anzuwendende Schema definiert werden kann.

Er ist wie folgt aufgebaut:

```
CUrl::CUrl(const string &url, Scheme s)
: m_url(url), m_scheme(UNDEFINED) {
divideUrl(s);
}
```

Listing 327: Ein Konstruktor von CUrl

divideUrl

Die Methode `divideUrl` zerlegt dann die als String vorliegende URL in ihre Einzelteile. Bevor wir `divideUrl` genauer betrachten, werfen wir zunächst einen Blick auf die URL-Definition in RFC1738:

```
url             = httpurl | ftpurl | newsurl |
                  nntpurl | telneturl | gopherurl |
                  waisurl | mailtourl | fileurl |
                  prosperourl | otherurl
```

Wir haben es hier wieder mit der bereits in Rezept 18 eingesetzten Backus-Naur-Form (BNF) zu tun. Allerdings wurde hier die Syntax etwas abgeändert. Anstelle von `::=` wird nur das Gleichheitszeichen verwendet. Alle aus mehreren Buchstaben bestehenden Wörter sind nicht-terminale Symbole. Sie stehen hier nicht in spitzen Klammern <>. Tatsächliche Zeichen stehen in doppelten Anführungszeichen. Das Zeichen | definiert Alternativen.

Die für uns interessanten URL-Definitionen sind `httpurl`, `ftpurl` und `fileurl`.

```
ftpurl          = "ftp://" login [ "/" fpath [ ";type=" ftptype ]]
httpurl         = "http://" hostport [ "/" hpath [ "?" search ]]
fileurl         = "file://" [ host | "localhost" ] "/" fpath
```

Obwohl die Definition für `httpurl` keinen Login zulässt, ist dies gängige Praxis. Häufig liegen HTML-Dokumente in mit Passwort geschützten Bereichen. Deswegen ändern wir für unsere Klasse die Definition von `httpurl` folgendermaßen ab:

```
httpurl         = "http://" login [ "/" hpath [ "?" search ]]
```

Wir sehen, dass zu Beginn immer der Name des Schemas steht. Darüber werden wir später bei einer absoluten URL bestimmen, nach welchen Schema sie zerlegt werden soll.

Hier ist `divideUrl`, die Erklärung folgt hinter dem Listing:

```
bool CUrl::divideUrl(Scheme s) {
  insensitive_string url=m_url.c_str();
  size_t be=0;

/*
** Schema isolieren
*/
  be=scheme(url, be);

/*
** Schema ist HTTP?
** => Login, Pfad, Parameter und Fragment isolieren
*/
  if(m_scheme==HTTP) {
    be=login(url, be);
    be=httppath(url, be);
    be=search(url, be);
    be=fragment(url, be);
  }

/*
** Schema ist FTP?
** => Pfad und Type isolieren
*/
  else if(m_scheme==FTP) {
    be=login(url, be);
    be=ftppath(url, be);
    be=ftptype(url, be);
  }

/*
** Schema ist FILE?
** => Host und Pfad isolieren
*/
  else if(m_scheme==FILE) {
```

Listing 328: Die Methode divideUrl

```
    be=filehost(url, be);
    be=filepath(url, be);
  }

/*
** Partielle URL?
*/
  else if(m_scheme==PARTIAL) {
    if(be<url.length()) {
      if(url[be]!='#') {

/*
** Auftreten von // oder / am Anfang des Pfades
** isolieren
*/
        if(url.substr(be,2)=="//") {
          m_folders.push_back("//");
          be+=2;
        }
        else if(url.substr(be,1)=="/") {
          m_folders.push_back("/");
          be+=1;
        }
      }
    }

/*
** Soll partielle URL als HTTP interpretiert werden?
** => Relativen Pfad, Parameter und Fragment isolieren
*/
    if(s==HTTP) {
      be=httppath(url, be);
      be=search(url, be);
      be=fragment(url, be);
    }

/*
** Soll partielle URL als FTP interpretiert werden?
** => Relativen Pfad und Type
*/
    else if(s==FTP) {
```

Listing 328: Die Methode divideUrl (Forts.)

```
      be=ftppath(url, be);
      be=ftptype(url, be);
    }

/*
** Soll partielle URL als FILE interpretiert werden?
** => Relativen Pfad isolieren
*/
    else if(s==FILE) {
      be=filepath(url, be);
    }
  }
  return(true);
}
```

Listing 328: Die Methode divideUrl (Forts.)

Zuerst wird durch den Aufruf von `scheme` das Schema der URL bestimmt. Entsprechend der BNF für die einzelnen Schemata werden dann verschiedene Methoden aufgerufen, die die Url zerlegen.

Ein besonderer Fall muss bei partiellen URLs berücksichtigt werden. Obwohl das Zeichen `/` zum Trennen der einzelnen Pfadteile eingesetzt wird, besitzt ein einfacher oder zweifacher (`//`) Slash am Anfang einer partiellen URL eine besondere Bedeutung und wird deswegen mit in die Ordner-Liste aufgenommen.

scheme

Bevor wir weiter in die Details der URL-Zerlegung einsteigen, folgt hier erst einmal die Methode `scheme`:

```
size_t CUrl::scheme(const insensitive_string &url,
                    size_t be) {

/*
** Bei nicht-partieller URL Schema isolieren
** und auswerten
*/
  if(isUrlPartial(m_url)) {
    m_scheme=PARTIAL;
    return(be);
```

Listing 329: Die Methode scheme

```
  }

  if(url.find("http://",be)==be) {
    m_scheme=HTTP;
    m_protocol=url.substr(0,7).c_str();
    return(be+7);
  }

  if(url.find("file://",be)==be) {
    m_scheme=FILE;
    m_protocol=url.substr(0,7).c_str();
    return(be+7);
  }

  if(url.find("ftp://",be)==be) {
    m_scheme=FTP;
    m_protocol=url.substr(0,6).c_str();
    return(be+6);
  }
  throw(EUrlError());
  return(be);
}
```

Listing 329: Die Methode scheme (Forts.)

isUrlPartial

Es ist vielleicht noch interessant zu erfahren, wie die Methode `isUrlPartial` feststellt, ob es sich um eine partielle URL handelt oder nicht. Per Definition ist jede URL nicht partiell, bei der irgendwo vor dem ersten Slash ein Doppelpunkt vorhanden ist. Umgesetzt sieht das so aus:

```
bool CUrl::isUrlPartial(const std::string &url) {

/*
** Bei einer nicht-partiellen URL müssen die Zeichen
** Doppelpunkt und Slash vorhanden sein und der erste
** Doppelpunkt muss vor dem ersten Slash stehen.
*/
  size_t colonpos=url.find(":");
  size_t slashpos=url.find("/");
```

Listing 330: Die Methode isUrlPartial

```
  if((colonpos!=string::npos)&&
    (slashpos!=string::npos)&&
     (colonpos<slashpos))
     return(false);
  else
    return(true);
}
```

Listing 330: Die Methode isUrlPartial (Forts.)

http-URL

Befassen wir uns zunächst mit der Zerlegung einer http-URL. Dazu wollen wir folgende Definition benutzen:

```
httpurl        = "http://" login [ "/" hpath [ "?" search ]]
```

Umgesetzt im Quellcode sah das bei `divideUrl` so aus:

```
    be=login(url, be);
    be=httppath(url, be);
    be=search(url, be);
    be=fragment(url, be);
```

Wobei die Methode `httppath` dem `hpath` in der BNF entspricht. Als Ergänzung finden wir im Quellcode noch die Methode `fragment`, die einen mit # angehängten Fragment-Bezeichner isoliert. Fragment-Bezeichner werden bei http-URLs eingesetzt, um beispielsweise eine Position innerhalb eines HTML-Dokuments anzugeben:

```
http://www.addison-wesley.de/Codebook/Willms/content.html#chapter5
```

login

Wenden wir uns dem Login zu. In der BNF ist er so definiert:

```
login          = [ user [ ":" password ] "@" ] hostport
```

Das findet sich auch in der Methode `login` wieder:

```
size_t CUrl::login(const insensitive_string &url,
                   size_t be) {
  if(m_scheme==PARTIAL)
    return(be);

/*
** Falls vorhanden, Username und Passwort isolieren
*/
  be=user(url, be);

/*
** Host und Port isolieren
*/
  return(hostport(url, be));
}
```

Listing 331: Die Methode login

Eine partielle URL besitzt keinen Login, trotzdem ist die Abfrage hier in `login` eigentlich unnötig, weil beim Aufruf in `divideUrl` bereits darauf geprüft wird. Vielleicht wird die Methode aber noch mal von anderer Stelle aufgerufen, sodass die Abfrage berechtigt ist.

user

Es fällt vielleicht auf, dass ein Aufruf von `password` fehlt. Da in der BNF ein Passwort von der Existenz eines Usernamens abhängig ist, liegt dieser Aufruf in der `user`-Methode:

```
size_t CUrl::user(const insensitive_string &url,
                  size_t be) {
  size_t colonpos=url.find(":", be);
  size_t atpos=url.find("@", be);
  size_t slashpos=url.find("/", be);
  size_t en;

/*
** Existiert ein @ und steht vor einem eventuellen / ?
```

Listing 332: Die Methode user

```
*/
  if((atpos!=insensitive_string::npos)&&(atpos<slashpos))

/*
** Existiert vor dem @ noch ein : ?
*/
    if((colonpos!=insensitive_string::npos)&&(colonpos<atpos))
      en=colonpos;
    else
      en=atpos;

/*
** Kein Klammeraffe vor einem / ?
** => Kein Username!
*/
  else
    return(be);

/*
** Username wird isoliert
*/
  m_username=url.substr(be,en-be).c_str();

/*
** Eventuelles Passwort wird isoliert
*/
  return(password(url, en));
}
```

Listing 332: Die Methode user (Forts.)

Der Username wird entweder mit einem : beendet (falls ein Passwort folgt) oder mit einem @ (falls kein Passwort folgt.) Dabei muss sichergestellt werden, dass sich das gefundene : oder @ vor einem Slash befindet.

password

Die Methode `user` ruft dann `password` auf:

```
size_t CUrl::password(const insensitive_string &url,
                      size_t be) {
```

Listing 333: Die Methode password

```
/*
** Aktuelles Zeichen ein @?
** => Nur Username, kein Passwort
*/
  if(url[be]=='@')
    return(be+1);

/*
** Passwort muss hinter einem : stehen
*/
  if(url[be]!=':')
    throw(EUrlError());

/*
** Passwort muss vor einem @ stehen
*/
  size_t atpos=url.find("@", be);
  size_t slashpos=url.find("/", be);
  size_t en;

  if((atpos!=insensitive_string::npos)&&(atpos<slashpos))
    en=atpos;
  else
    throw(EUrlError());

  be++;

/*
** Passwort isolieren
*/
  m_password=url.substr(be,en-be).c_str();
  return(en+1);
}
```

Listing 333: Die Methode password (Forts.)

Sollte vor dem vermeintlichen Passwort ein `@` stehen, dann existiert in Wirklichkeit kein Passwort. Wenn an der aktuellen Stelle weder ein `@` (Kein Passwort) noch ein `:` (Passwort vorhanden) steht, dann ist die URL fehlerhaft.

Wenn ein Passwort existiert, dann muss es auf jeden Fall mit einem `@` enden.

hostport

Den letzten Teil von `login` erledigt `hostport`:

```
hostport       = host [ ":" port ]
```

Als Methode stellt sich `hostport` folgendermaßen dar:

```
size_t CUrl::hostport(const insensitive_string &url,
                      size_t be) {
  return(host(url, be));
}
```

Listing 334: Die Methode hostport

Die Methode `hostport` ruft einfach `host` auf. Weil laut BNF ein Port nicht ohne Host definiert werden kann, wurde der Aufruf von `port` in die Methode `host` verlagert:

```
size_t CUrl::host(const insensitive_string &url,
                      size_t be) {

/*
** Leerer Host?
*/
  if(url[be]=='/')
    return(be+1);

/*
** / oder : oder Stringende markieren Ende des Hosts
*/
  size_t slashpos=url.find("/", be);
  size_t colonpos=url.find(":", be);
  size_t en;

/*
** Existiert ein : und steht dieser vor einem / ?
** => Hostende an :-Position
*/
  if((colonpos!=insensitive_string::npos)&&(colonpos<slashpos))
    en=colonpos;
```

```
/*
** Existiert ein / ?
** => Hostende an /-Position
*/
  else if(slashpos!=insensitive_string::npos)
    en=slashpos;

/*
** Kein / ?
** => Restlicher String ist Host
*/
  else
    en=url.length();

/*
** Host isolieren
*/
  m_domain=url.substr(be, en-be).c_str();

/*
** Eventuellen Port isolieren
*/
  return(port(url, en));
}
```

Der Host kann auf drei verschiedene Arten enden:

- Mit einem :, wenn noch ein Port folgt.
- Mit einem /, wenn direkt der Pfad folgt.
- Mit dem Stringende, falls hinter dem Host nichts mehr steht.

port

Ist der Host isoliert, wird `port` aufgerufen:

```
size_t CUrl::port(const insensitive_string &url,
                  size_t be) {

/*
```

Listing 335: Die Methode port

```
** Port steht hinter einem :
*/
  if(url[be]!=':')
    return(be+1);
  size_t slashpos=url.find("/", be);
  size_t en;

/*
** Existiert ein / ?
** => Port endet an /-Position
*/
  if(slashpos!=insensitive_string::npos)
    en=slashpos;

/*
** Kein / ?
** => Port endet am Stringende
*/
  else
    en=url.length();

  be++;

/*
** Port isolieren
*/
  m_port=url.substr(be, en-be).c_str();
  return(en+1);
}
```

Listing 335: Die Methode port (Forts.)

Der Port wiederum kann entweder mit dem Stringende oder mit einem / enden.

An dieser Stelle ist der Login komplett bearbeitet.

```
hpath          = hsegment *[ "/" hsegment ]
```

Die eckigen Klammern bedeuten, dass der eingeschlossene Ausdruck optional ist. Der Stern davor spezifiziert eine Hüllenbildung: Der Ausdruck dahinter kann beliebig oft aneinander gereiht werden.

Unter `hsegment` verstehen wir einen Teil des Pfades, der von anderen Pfad-Teilen mit `/` getrennt wird. In rfc1738 wird noch genauer spezifiziert, aus welchen Zeichen `hsegment` bestehen darf, aber das kann bei Interesse auf der CD nachgelesen werden.

httppath

Dem `hpath` in der BNF entspricht die Methode `httppath` in der Klasse `CUrl`:

```
size_t CUrl::httppath(const insensitive_string &url,
                      size_t be)  {
  if(be>=url.length())
    return(be);

/*
** Pfadteil endet mit /, ?, # oder Stringende
*/
  size_t slashpos=url.find("/", be);
  size_t questionpos=url.find("?", be);
  size_t gatepos=url.find("#", be);

/*
** Ermitteln, ob ? oder # zuerst kommt
*/
  size_t firstpos=(questionpos<=gatepos)?questionpos:gatepos;
  size_t en;

/*
** # oder ? vor Slash?
** => Pfadteil endet bei # oder ?
*/
  if((firstpos!=insensitive_string::npos)&&(firstpos<slashpos))
    en=firstpos;

/*
** Existiert ein / ?
** => Pfadteil endet bei /
*/
  else if(slashpos!=insensitive_string::npos)
    en=slashpos;

/*
** Kein / ?
```

Listing 336: Die Methode httppath

```
** => Pfadteil endet am Stringende
*/
  else
    en=url.length();


/*
** Pfadteil isolieren
*/
  m_folders.push_back(url.substr(be, en-be).c_str());

/*
** Pfadteil endet mit ? oder # ?
** => Pfad zu Ende
*/
  if((en==questionpos)||(en==gatepos))
    return(en);

/*
** Nächsten Pfadteil isolieren
*/
  return(httppath(url, en+1));
}
```

Listing 336: Die Methode httppath (Forts.)

Bei den üblichen Dateisystemen gibt es Ordner und Dateien. Es ist interessant, dass die URL-Definition diese Unterscheidung nicht vornimmt. An der bloßen URL lässt sich nicht erkennen, ob der letzte Pfad-Teil eine Datei oder einen Ordner darstellt, denn schließlich könnte man auch einen Ordner `index.htm` nennen. Letztlich weiß nur der Server, der auf die Dateistruktur zugreifen kann, ob es eine Datei oder ein Ordner ist.

Aus diesem Grund versucht unsere Klasse `CUrl` erst gar nicht, eine solche Unterscheidung zu treffen.

Damit kann ein Pfad-Teil einer http-URL auf folgende Arten enden:

- Mit einem `/`, falls noch weitere Teil-Pfade folgen.
- Mit einem `?`, falls Parameter folgen.
- Mit einem `#`, falls ein Fragment-Bezeichner folgt.
- Mit dem Stringende, falls nichts folgt.

Die Methode `httppath` ruft sich dann selbst rekursiv auf, um weitere Pfad-Teile zu isolieren.

search

Nachdem der Pfad zerlegt wurde, muss noch auf Parameter geprüft werden. Das erledigt die Methode `search`:

```
size_t CUrl::search(const insensitive_string &url,
                    size_t be)  {

/*
** Parameter stehen hinter einem ?
*/
  if(url[be]!='?')
    return(be);
  size_t gatepos=url.find("#", be);
  size_t en;

/*
** # vorhanden?
** => Parameter enden an #-Position
*/
  if(gatepos!=insensitive_string::npos)
    en=gatepos;

/*
** Kein # ?
** => Parameter enden am Stringende
*/
  else
    en=url.length();

/*
** Parameter isolieren
*/
  be++;
  m_parameters=url.substr(be, en-be).c_str();
  return(en);
}
```

Listing 337: Die Methode search

fragment

Zu guter Letzt kann hinter den Parametern noch ein Fragment-Bezeichner stehen, der dann von der Methode `fragment` isoliert wird:

```
size_t CUrl::fragment(const insensitive_string &url,
                      size_t be)  {

/*
** Fragment beginnt mit #
*/
  if(url[be]!='#')
    return(be);
  size_t en=url.length();

/*
** Fragment isolieren
*/
  be++;
  m_fragment=url.substr(be, en-be).c_str();
  return(en);
}
```

Listing 338: Die Methode fragment

file-URL

Eine file-URL ist ziemlich einfach aufgebaut:

```
fileurl        = "file://" [ host | "localhost" ] "/" fpath
```

Eine file-URL besitzt keinen Login. Sie beginnt direkt mit dem Host oder der Zeichenkette »localhost«. Im Programm ist allerdings keine Unterscheidung notwendig, weil der Host selbst von uns nicht weiter zerlegt wird.

Wir haben aber mit der bisher implementierten `host`-Methode ein grundsätzlicheres Problem: Bei http- und ftp-URLs dient der Doppelpunkt beim Host der Abgrenzung des Ports. Bei einer file-URL darf der Doppelpunkt jedoch als Zeichen im Host enthalten sein.

filehost

Deswegen müssen wir hier speziell eine eigene Methode implementieren, die `filehost` heißen soll:

```
size_t CUrl::filehost(const insensitive_string &url,
                      size_t be) {

/*
** Leerer Host?
*/
  if(url[be]=='/')
    return(be+1);

/*
** / oder Stringende markieren Ende des Hosts
*/
  size_t slashpos=url.find("/", be);
  size_t en;

/*
** Existiert ein / ?
** => Hostende an /-Position
*/
  if(slashpos!=insensitive_string::npos)
    en=slashpos;

/*
** Kein / ?
** => Restlicher String ist Host.
*/
  else
    en=url.length();

/*
** Host isolieren
*/
  m_domain=url.substr(be, en-be).c_str();

/*
** Eventuellen Port isolieren
*/
  return(en);
}
```

Listing 339: Die Methode filehost

filepath

`fpath` ist in der BNF wie folgt definiert:

```
fpath          = fsegment *[ "/" fsegment ]
```

Der Unterschied zwischen `fsegment` und `hsegment` liegt darin, dass bei einem `fsegment` auch das Zeichen `?` benutzt werden darf. Bei einer http-URL wurde das Fragezeichen zum Trennen zwischen Pfad und Parametern benötigt.

Wir können daher die Methode `httppath` nicht verwenden und müssen eine neue Methode `filepath` implementieren:

```
size_t CUrl::filepath(const insensitive_string &url,
                      size_t be)  {
  if(be>=url.length())
    return(be);

/*
** Pfadteil endet mit / oder Stringende
*/
  size_t slashpos=url.find("/", be);
  size_t en;

/*
** / vorhanden?
** => Pfadteil endet an /-Position
*/
  if(slashpos!=insensitive_string::npos)
    en=slashpos;

/*
** Kein / ?
** => Pfadteil endet am Stringende
*/
  else
    en=url.length();

/*
** Pfadteil isolieren
```

Listing 340: Die Methode filepath

```
*/
  m_folders.push_back(url.substr(be, en-be).c_str());

/*
** Eventuell weiteren Pfadteil isolieren
*/
  return(filepath(url, en+1));
}
```

Listing 340: Die Methode filepath (Forts.)

ftp-URL

Zur Erinnerung noch einmal der Aufbau einer ftp-URL:

```
ftpurl         = "ftp://" login [ "/" fpath [ ";type=" ftptype ]]
```

ftppath

Der Login ist identisch mit dem der http-URL. Aber `fpath` bentötigt zusätzliche Aufmerksamkeit, weil jetzt hinter dem Pfad, mit Semikolon abgetrennt, der FTP-Type stehen kann. Deswegen hier die Methode `ftppath`:

```
size_t CUrl::ftppath(const insensitive_string &url,
                     size_t be)  {
  if(be>=url.length())
    return(be);

/*
** Pfadteil endet mit / oder ; oder Stringende
*/
  size_t slashpos=url.find("/", be);
  size_t semicolonpos=url.find(";", be);

/*
** Ermitteln, ob / oder ; zuerst kommt
*/
  size_t firstpos=(slashpos<=semicolonpos)?slashpos:semicolonpos;
  size_t en;
```

Listing 341: Die Methode ftppath

```
/*
** / oder ; ?
** => Pfadteil endet bei / oder ;
*/
  if(firstpos!=insensitive_string::npos)
    en=firstpos;

/*
** Kein / oder ; ?
** => Pfadteil endet am Stringende
*/
  else
    en=url.length();

/*
** Pfadteil isolieren
*/
  m_folders.push_back(url.substr(be, en-be).c_str());

/*
** Pfadteil endete mit ; ?
** => Pfad zu Ende
*/
  if(en==semicolonpos)
    return(en);

/*
** Nächsten Pfadteil isolieren
*/
  return(ftppath(url, en+1));
}
```

Listing 341: Die Methode ftppath (Forts.)

ftptype

Für das ordnungsgemäße Zerlegen fehlt nun nur noch die Methode `ftptype`:

```
size_t CUrl::ftptype(const insensitive_string &url,
                     size_t be)  {
```

Listing 342: Die Methode ftptype

```
/*
** ftptype beginnt mit ;
*/
  if(url[be]!=';')
    return(be);
  size_t en=url.length();
  size_t equalpos=url.find("=",be);

/*
** Existiert = ?
** => ftptype hinter = isolieren
*/
  if(equalpos!=insensitive_string::npos) {
    m_ftptype=url.substr(equalpos+1).c_str();
    return(en);
  }
  return(be);
}
```

Listing 342: Die Methode ftptype (Forts.)

showUrl

Damit die Zerlegung auch problemlos betrachtet werden kann, wurde die Methode showUrl implementiert:

```
void CUrl::showUrl() const{
  cout << "URL        :" << m_url << endl;
  cout << "Protokoll  :" << m_protocol << endl;
  cout << "Username   :" << m_username << endl;
  cout << "Passwort   :" << m_password << endl;
  cout << "Domaene    :" << m_domain << endl;
  cout << "Port       :" << m_port << endl;
  cout << "Ordner     :" << endl;
  for(size_t i=0; i<m_folders.size(); i++)
    cout << "             " << m_folders[i] << endl;
  cout << "Parameter  :" << m_parameters << endl;
  cout << "Fragment   :" << m_fragment << endl;
  cout << "FTP-Type   :" << m_ftptype << endl;
}
```

Listing 343: Die Methode showUrl

Die Klasse `CUrl` finden Sie auf der CD in den `CUrl`-Dateien.

65 Wie kann eine partielle URL aufgelöst werden?

Das Besondere an einer partiellen URL liegt in ihrem relativen Charakter. Während eine absolute URL den kompletten Pfad samt Server zu einer Datei beinhaltet, bezieht sich die partielle URL auf einen bereits adressierten Ort.

Eine partielle URL kann somit nur aufgelöst werden, wenn eine absolute URL als Bezugspunkt vorhanden ist.

Wir wollen die Klasse `CUrl` aus Rezept 64 so erweitern, dass sie den neuen Ansprüchen gerecht wird. `CUrl` ist in ihrer aktuellen Implementierung bereits in der Lage, eine partielle URL zu zerlegen. Es fehlt nur noch die Funktionaliät, eine partielle URL in einen Bezug zu einer absoluten URL zu setzen.

Konstruktor

Dazu wird `CUrl` um einen Konstruktor ergänzt:

```
CUrl::CUrl(const string &url, const string &relurl)
: m_url(url), m_scheme(UNDEFINED) {

/*
** Erste URL muss eine vollständige und die zweite eine
** partielle sein.
*/
  if(isUrlPartial(url)||!isUrlPartial(relurl))
    throw(EUrlError());

/*
** Umwandeln der URLs in CUrl-Objekte und Aufruf
** von combineUrls
*/
  CUrl curl(url), crelurl(relurl,curl.m_scheme);
  *this=combineUrls(curl, crelurl);
}
```

Listing 344: Ein zusätzlicher Konstruktor für CUrl

Dem neuen Konstruktor werden die Basis-URL und die partielle URL als String übergeben. Zunächst wird sichergestellt, dass es sich bei der ersten URL tatsächlich um eine absolute und bei der zweiten URL um eine partielle URL handelt.

Dann werden die beiden URLs in `CUrl`-Objekte umgewandelt. Dabei wird der partiellen URL gleich das Schema der Basis-URL mitgegeben.

combineUrls

Anschließend wird die Methode `combineUrls` aufgerufen, der eine Basis-URL und eine partielle URL als `CUrl`-Objekt übergeben werden und die dann das Ergebnis wieder als `CUrl`-Objekt zurückgibt. Hier als Erstes das Listing der Methode:

```
CUrl CUrl::combineUrls(const CUrl &base, const CUrl &rel) {
  CUrl result;
  folder_container::const_iterator riter=rel.m_folders.begin();

/*
** Existieren keine Teilpfade?
** => Basis-URL mit Parametern und Fragment von
** partieller URL ergänzen
*/
  if((base.m_scheme==HTTP)&&(rel.m_folders.size()==0)) {
    result=base;
    result.m_parameters=rel.m_parameters;
    result.m_fragment=rel.m_fragment;
    result.buildUrl();
    return(result);
  }

/*
** Partielle URL beginnt mit // ?
** => Der gesamte Pfad der Basis-URL, selbst der Host,
**    wird gelöscht.
*/
  if(rel.m_folders.front()=="//") {
    result.m_url=base.m_protocol+rel.m_url.substr(2);
    result.divideUrl(base.m_scheme);
    return(result);
  }

/*
** Außer den Parametern und dem Fragment werden
** alle Bestandteile der Basis-URL übernommen.
*/
  result=base;
  result.m_parameters=rel.m_parameters;
  result.m_fragment=rel.m_fragment;
  result.m_ftptype=rel.m_ftptype;
```

Listing 345: Die Methode combineUrls

```
/*
** Beginnt URL mit / ?
** => der gesamte Pfad wird gelöscht.
*/
  if(rel.m_folders.front()=="/") {
    result.m_folders.clear();
    riter+=1;
  }

/*
** Andernfalls wird der letzte Teil des Pfades
** (Üblicherweise der Dateiname) gelöscht.
*/
  else
    result.m_folders.pop_back();

/*
** Alle Pfad-Teile (den letzten ausgenommen)
** werden durchlaufen
*/
  bool basereached=(result.m_folders.size()==0);
  for(;riter!=rel.m_folders.end(); ++riter) {

/*
** Pfadteil in partieller URL gleich .. und
** Ergebnis-Url hat noch Pfadteile?
** => Der aktuell letzte Pfadteil wird gelöscht.
*/
    if((!basereached)&&(*riter="..")) {
      result.m_folders.pop_back();
      if(result.m_folders.size()==0)
        basereached=true;
    }

/*
** Pfadteil in partieller URL gleich . ?
** => Ignorieren
*/
    else if(*riter!=".")
      result.m_folders.push_back(*riter);
```

Listing 345: Die Methode combineUrls (Forts.)

```
  }

/*
** Aus den Einzelteilen wird die Gesamt-URL
** aufgebaut.
*/
  result.buildUrl();
  return(result);
}
```

Listing 345: Die Methode combineUrls (Forts.)

Um die Besonderheiten beim Auflösen einer partiellen URL genauer zu betrachten, wollen wir folgende fiktive URL als Basis-URL annehmen:

```
http://www.addison-wesley.de/Codebook/Willms/content.html
```

Zu Beginn der partiellen URL geht es mit den Ausnahmen bereits los. Sollte die partielle URL mit `//` beginnen, dann wird der komplette Pfad samt Server aus der Basis-URL gelöscht. Die partielle URL

```
//x/y.html
```

würde bezogen auf unsere Basis-URL folgendermaßen aufgelöst:

```
http://x/y.html
```

Beginnt die partielle URL lediglich mit einem Slash, dann bleibt der Server erhalten:

```
/x/y.html
```

ergibt

```
http://www.addison-wesley.de/x/y.html
```

Am einfachsten sind die partiellen URLs ohne führende Slashs umzusetzen. Bei ihnen wird aus der Basis-URL lediglich der letzte Pfad-Teil entfernt. Damit ergibt

```
x/y.html
```

die URL

```
http://www.addison-wesley.de/Codebook/Willms/x/y.html
```

Eine weitere Besonderheit kommt dem Pfadteil .. zugute. Er sorgt dafür, dass in der Pfad-Hierarchie eine Ebene nach oben gewechselt wird, was in der Basis-URL dem Löschen des aktuell letzten Pfadteils gleichkommt. Zwei Beispiele:

```
../x/y.html
```

ergibt

```
http://www.addison-wesley.de/Codebook/x/y.html
```

und

```
../../x/y.html
```

löst sich auf zu

```
http://www.addison-wesley.de/x/y.html
```

Sollte .. jedoch in Kombination mit anderen Zeichen auftreten, dann wird es als Namensbestandteil aufgefasst:

```
../..w/x/y.html
```

ist damit

```
http://www.addison-wesley.de/Codebook/..w/x/y.html
```

Sollte es mehr Vorkommen von .. geben, als die Basis-URL Ordner-Ebenen besitzt, dann gelten die restlichen .. als Pfadteile:

```
../../../../x/y.html
```

wird aufgelöst zu

```
http://www.addison-wesley.de/../../x/y.html
```

.. kann auch mitten im Pfad vorkommen:

```
../w/x/../y.html
```

ergibt

```
http://www.addison-wesley.de/Codebook/w/y.html
```

Der einzelne Punkt steht für das aktuelle Verzeichnis der Basis-URL und kann bei der Auflösung ignoriert werden:

```
.././x/././y.html
```

löst sich auf zu

```
http://www.addison-wesley.de/Codebook/x/y.html
```

Zum Schluss muss bei http-URLs noch darauf geachtet werden, dass eine partielle URL nur aus einem Fragment-Bezeichner bestehen kann:

```
#chapter2
```

Dieser Fragment-Bezeichner wird dann einfach an die Basis-URL angehängt:

```
http://www.addison-wesley.de/Codebook/Willms/content.html#chapter2
```

buildUrl

Wenn die partielle URL anhand dieser Regeln aufgelöst wurde, müssen danach die Einzelteile zu einer Gesamt-URL zusammengesetzt werden.

Diese Aufgabe erledigt die Methode `buildUrl`:

```
void CUrl::buildUrl() {

/*
** Schema hinzufügen
*/
  m_url=m_protocol;

/*
** Bei HTTP und FTP Username und Passwort
** berücksichtigen, falls vorhanden
*/
  if((m_scheme==HTTP)||(m_scheme==FTP)) {
    if(m_username!="") {
      m_url+=m_username;
      if(m_password!="")
        m_url+=":"+m_username;
      m_url+="@";
    }
  }

/*
** Host hinzufügen
```

Listing 346: Die Methode buildUrl

```
*/
  m_url+=m_domain;

/*
** Bei HTTP und FTP Port
** berücksichtigen, falls vorhanden
*/
  if((m_scheme==HTTP)||(m_scheme==FTP)) {
    if(m_port!="")
      m_url+=":"+m_port;
  }

/*
** Alle Teilpfade hinzufügen
*/
  for(folder_container::const_iterator iter=m_folders.begin();
      iter!=m_folders.end();
       ++iter)
    m_url+="/"+*iter;

/*
** Bei HTTP Parameter und Fragment-Bezeichner
** berücksichtigen, falls vorhanden
*/
  if(m_scheme==HTTP) {
    if(m_parameters!="")
      m_url+="?"+m_parameters;
    if(m_fragment!="")
      m_url+="#"+m_fragment;
  }

/*
** Bei FTP den FTP-Type
** berücksichtigen, falls vorhanden
*/
  if(m_scheme==FTP) {
    if(m_ftptype!="")
      m_url+=";type="+m_ftptype;
  }
}
```

Listing 346: Die Methode buildUrl (Forts.)

Die Klasse `CUrl` finden Sie auf der CD in den `CUrl`-Dateien.

66 Wie können die Parameter einer http-URL zerlegt werden?

Dazu greifen wir auf die `CUrl`-Klasse aus dem vorherigen Rezept zurück. `CUrl` hat die URL bereits so weit aufgespaltet, dass die Parameter in einem String namens `m_parameters` vorliegen.

Wir gehen nun noch einen Schritt weiter und verwenden für `m_parameters` einen erweiterten Datentyp:

```
typedef std::pair<std::string, std::string> parameter_type;
typedef std::vector<parameter_type> parameter_container;
parameter_container m_parameters;
```

Die Parameter liegen nun in einem Vektor paarweise in Parametername und Parameterwert aufgeteilt.

search

Dazu müssen innerhalb von `CUrl` einige Methoden angepasst werden. Allen voran natürlich `search`, die für das Isolieren der Parameter verantwortlich ist:

```
size_t CUrl::search(const insensitive_string &url,
                    size_t be)  {

/*
** Parameter stehen hinter einem ?
*/
  if(url[be]!='?')
    return(be);
  size_t gatepos=url.find("#", be);
  size_t en;

/*
** # vorhanden?
** => Parameter enden an #-Position
*/
  if(gatepos!=insensitive_string::npos)
    en=gatepos;

/*
```

Listing 347: Die angepasste Methode search

```
** Kein # ?
** => Parameter enden am Stringende
*/
  else
    en=url.length();

/*
** Parameter isolieren
*/
  be++;
  string parameters=url.substr(be, en-be).c_str();
  size_t pbe=0;
  while(pbe<parameters.length()) {

/*
** Ende des aktuellen Parameters bestimmen
*/
    size_t pen=parameters.find("&",pbe);
    if(pen==string::npos)
      pen=parameters.length();

/*
** Aktuellen Parameter isolieren
*/
    string param=parameters.substr(pbe,pen-pbe);

/*
** Prüfen, ob es Parameter oder Wert ist
*/
    size_t equalpos=param.find("=");
    if(equalpos!=string::npos)
      m_parameters.push_back(make_pair(param.substr(0,equalpos),
                                       param.substr(equalpos+1)));
    else
      m_parameters.push_back(make_pair("",param));
    pbe=pen+1;
  }
  return(en);
}
```

Listing 347: Die angepasste Methode search (Forts.)

Bei der Verwaltung der Parameter stehen wir vor dem Problem, zwischen Parametern zu unterscheiden, die keinen Wert besitzen (z.B: `?Name=`), und Parametern, die nur aus einem Wert bestehen (z.B. `?nospace`).

Wir lösen das Problem, indem die Parameter ohne Wert als Parameterwert einen leeren String bekommen und die nur aus Werten bestehenden Parameter einen leeren String als Parameternamen.

getParameters

Die Methode `getParameters` muss dem neuen Datentyp entsprechend angepasst werden. Sie wird als inline direkt in der Klassendefinition definiert:

```
const parameter_container &getParameters() const {
    return(m_parameters);
  }
```

Listing 348: Die angepasste Methode getParameters

getParametersAsString

Um die Parameter gleich in einem String zur Verfügung zu haben, fügen wir die Methode `getParametersAsString`, die die Parameter mit & getrennt – allerding ohne führendes ? – als String liefert:

```
string CUrl::getParametersAsString() const {
  string params;

  if(m_scheme==HTTP) {
  if(m_parameters.size()!=0)
    for(parameter_container::const_iterator piter=
                                  m_parameters.begin();
        piter!=m_parameters.end();
        ++piter) {

/*
** Die Parameterpaare mit & trennen
*/
      if(piter!=m_parameters.begin())
        params+="&";
```

Listing 349: Die Methode getParametersAsString von CUrl

```
/*
** Bloßer Parameter-Wert?
*/
      if(piter->first=="")
        params+=piter->second;

/*
** Parameter-Paar
*/
      else
        params+=piter->first+"="+piter->second;
    }
  }
  return(params);
}
```

Listing 349: Die Methode getParametersAsString von CUrl (Forts.)

isParamValid

Um die Parameter etwas eleganter ansprechen zu können, werden zwei weitere Methoden implementiert. Einmal wäre da `isParamValid`, die prüft, ob ein bestimmter Parametername vorhanden ist:

```
bool CUrl::isParamValid(const std::string &p) const {

/*
** Alle Parameterpaare durchlaufen
*/
  for(parameter_container::const_iterator iter=m_parameters.begin();
      iter!=m_parameters.end();
      ++iter)

/*
** Ist Parametername, oder bei leerem Parameternamen
** der Parameterwert der gesuchte?
*/
    if((iter->first==p)||((iter->first=="")&&(iter->second==p)))
      return(true);
  return(false);
}
```

Listing 350: Die Methode isParamValid

Die Methode sucht nach einer Übereinstimmung mit dem Parameternamen oder falls dieser leer sein sollte nach einer Übereinstimmung mit dem Parameterwert.

getParamValue

Um über den Parameternamen an den Parameterwert zu kommen, wurde die Methode `getParamValue` implementiert:

```
string CUrl::getParamValue(const std::string &p) const {

/*
** Alle Parameterpaare durchlaufen
*/
  for(parameter_container::const_iterator iter=m_parameters.begin();
      iter!=m_parameters.end();
      ++iter)

/*
** Ist Parametername, oder bei leerem Parameternamen
** der Parameterwert der gesuchte?
*/
    if((iter->first==p)||((iter->first=="")&&(iter->second==p)))
      return(iter->second);
  return("");
}
```

Listing 351: Die Methode getParamValue

buildUrl

Zum Schluss muss noch die Methode `buildUrl` aktualisiert werden, die aus den Einzelteilen wieder eine komplette URL erzeugt:

```
void CUrl::buildUrl() {

/*
** Schema hinzufügen
*/
  m_url=m_protocol;

/*
```

Listing 352: Die Methode buildUrl

```
** Bei HTTP und FTP Username und Passwort
** berücksichtigen, falls vorhanden
*/
  if((m_scheme==HTTP)||(m_scheme==FTP)) {
    if(m_username!="") {
      m_url+=m_username;
      if(m_password!="")
        m_url+=":"+m_username;
      m_url+="@";
    }
  }

/*
** Host hinzufügen
*/
  m_url+=m_domain;

/*
** Bei HTTP und FTP Port
** berücksichtigen, falls vorhanden
*/
  if((m_scheme==HTTP)||(m_scheme==FTP)) {
    if(m_port!="")
      m_url+=":"+m_port;
  }

/*
** Alle Teilpfade hinzufügen
*/
  for(folder_container::const_iterator iter=m_folders.begin();
      iter!=m_folders.end();
      ++iter)
    m_url+="/"+*iter;

/*
** Bei HTTP Parameter und Fragment-Bezeichner
** berücksichtigen, falls vorhanden
*/
  if(m_scheme==HTTP) {
    if(m_parameters.size()!=0)
      for(parameter_container::iterator piter=m_parameters.begin();
```

Listing 352: Die Methode buildUrl (Forts.)

```
          piter!=m_parameters.end();
          ++piter) {

/*
** Vor erstem Parameter-Paar muss ein ? stehen, vor
** den restlichen ein &
*/
        if(piter==m_parameters.begin())
          m_url+="?";
        else
          m_url+="&";

/*
** Bloßer Parameter-Wert?
*/
        if(piter->first=="")
          m_url+=piter->second;

/*
** Parameter-Paar
*/
        else
          m_url+=piter->first+"="+piter->second;
      }

    if(m_fragment!="")
      m_url+="#"+m_fragment;
  }

/*
** Bei FTP den FTP-Type
** berücksichtigen, falls vorhanden
*/
  if(m_scheme==FTP) {
    if(m_ftptype!="")
      m_url+=";type="+m_ftptype;
  }
}
```

Listing 352: Die Methode buildUrl (Forts.)

Die neue Methode `showUrl` ersparen wir uns hier.

Die Klasse `CUrl` finden Sie auf der CD in den `CUrl`-Dateien.

67 Wie wird eine URL korrekt kodiert?

Insbesondere bei Pfad- und Dateinamen können Zeichen vorkommen, die in einer URL eigentlich nicht erlaubt sind. Der folgende Dateiname ist streng genommen ungültig:

```
andré willms.html
```

Die Sonderzeichen (Leerzeichen und é) müssen speziell kodiert werden:

```
andr%e9%20willms.html
```

Der Doppelpunkt muss beispielsweise mit `%3a` kodiert werden. Allerdings dürfen Zeichen mit syntaktischer Natur nicht kodiert werden, wie beispielsweise der Doppelpunkt vor dem Port oder zwischen Username und Passwort.

Wir müssen die URL also in ihre Bestandteile zerlegen und diese dann ohne die syntaktischen Zeichen kodieren.

Diese Arbeit erledigt bereits die Klasse `CUrl` aus Rezept 1. Wir brauchen sie daher lediglich um die gewünschte Funktionalität zu erweitern.

encodeString

Wir implementieren zunächst eine allgemeine Methode `encodeString`, die einen beliebigen String korrekt kodiert. Weil diese Methode auch unabhängig von `CUrl` interessant sein kann, wollen wir sie als statische Methode deklarieren:

```
string CUrl::encodeString(const string &s) {
  string tmp;
  for(string::const_iterator iter=s.begin(); iter!=s.end(); ++iter)
    if(((*iter>=45)&&(*iter<=46))||
       ((*iter>=48)&&(*iter<=57))||
       ((*iter>=65)&&(*iter<=90))||
       ((*iter>=97)&&(*iter<=122))||
       (*iter==42)||(*iter==95))
       tmp+=*iter;
    else
```

Listing 353: Die statische Methode encodeString

```
      tmp+="%"+valueToHex((static_cast<int>(*iter)+256)%256);
  return(tmp);
}
```

Listing 353: Die statische Methode encodeString (Forts.)

`encodeString` macht von der Funktion `valueToHex` aus Rezept 27 Gebrauch.

encodeUrl

Nun fehlt noch eine Methode `encodeUrl`, die alle wesentlichen Teile der URL kodiert. Sie soll nicht von außerhalb aufrufbar sein, deswegen wird sie als privat deklariert:

```
void CUrl::encodeUrl() {
  for(folder_container::iterator fiter=m_folders.begin();
      fiter!=m_folders.end();
      ++fiter)
    *fiter=encodeString(*fiter);

  for(parameter_container::iterator piter=m_parameters.begin();
      piter!=m_parameters.end();
      ++piter) {
    piter->first=encodeString(piter->first);
    piter->second=encodeString(piter->second);
  }
}
```

Listing 354: Die Methode encodeUrl

getEncodedUrl

Um nun tatsächlich an die kodierte URL zu kommen, implementieren wir noch die Methode `getEncodedUrl`, die eine Kopie der URL anfertigt, diese dann kodiert und zurückgibt. Auf diese Weise bleibt das Original erhalten:

```
CUrl CUrl::getEncodedUrl() {
  CUrl tmp=*this;
  tmp.encodeUrl();
  tmp.buildUrl();
  return(tmp);
}
```

Listing 355: Die Methode getEncodedUrl

Zugriffsmethoden

Die Klasse `CUrl` hat bisher nicht die Möglichkeit, Änderungen an Teilen der URL vorzunehmen. Der Vollständigkeit halber wollen wir das jetzt nachholen. Zunächst die trivialen `set`-Methoden:

```
void setUrl(const std::string &s) {
  m_url=s;
  buildUrl();
}

void setUsername(const std::string &s) {
  m_username=s;
  buildUrl();
}

void setPassword(const std::string &s) {
  m_password=s;
  buildUrl();
}

void setDomain(const std::string &s) {
  m_domain=s;
  buildUrl();
}

void setPort(const std::string &s) {
  m_port=s;
  buildUrl();
}

void setFragment(const std::string &s) {
  m_fragment=s;
  buildUrl();
}

void setFtptype(const std::string &s) {
  m_ftptype=s;
  buildUrl();
}
```

Listing 356: Die set-Methoden

Jede `set`-Methode ruft nach der Änderung eines Attributs die Methode `buildUrl` auf, damit die komplette, über `getUrl` verfügbare URL aktualisiert wird.

addPath

Um den Pfad zu manipulieren, implementieren wir zwei Methoden `addPath` und `deletePath`, die einen neuen Pfadteil an den Pfad anhängen bzw. den letzten Pfadteil löschen:

```
void CUrl::addPath(const string &s) {
  m_folders.push_back(s);
  buildUrl();
}
```

Listing 357: Die Methode addPath von CUrl

deletePath

```
string CUrl::deletePath() {
  std::string tmp=m_folders.back();
  m_folders.pop_back();
  buildUrl();
  return(tmp);
}
```

Listing 358: Die Methode deletePath von CUrl

getPath

Mittels `getPath` erhalten wir den Pfad als String:

```
string CUrl::getPath() const {
  if(m_folders.size()==0)
    return("");

  folder_container::const_iterator iter=m_folders.begin();
  string path=*(iter++);

  while(iter!=m_folders.end())
    path+="/" + *(iter++);
```

Listing 359: Die Methode getPath von CUrl

```
  return(path);
}
```

Listing 359: Die Methode getPath von CUrl (Forts.)

setParam

Die Manipulation der Parameter erfordert etwas mehr Aufwand. Als Erstes wird eine Methode `setParam` definiert, mit deren Hilfe einem Parameter ein neuer Wert zugewiesen werden kann. Sollte der Parameter noch nicht vorhanden sein, so wird er angelegt. So ist gewährleistet, dass kein Parameter doppelt vorkommt.

```
void CUrl::setParam(const string &p, const string &v) {

/*
** Alle Parameterpaare durchlaufen
*/
  for(parameter_container::iterator iter=m_parameters.begin();
      iter!=m_parameters.end();
      ++iter)

/*
** Wird Parametername gefunden?
** => Parameter bekommt neuen Wert
*/
    if(iter->first==p) {
      iter->second=v;
      buildUrl();
      return;
    }

/*
** Parametername wurde nicht gefunden.
** => Neues Parameter-Paar wird angelegt.
*/
  m_parameters.push_back(make_pair(p,v));
  buildUrl();
}
```

Listing 360: Die Methode setParam von CUrl

deleteParam

Um ein Parameterpaar zu löschen, definieren wir die Methode `deleteParam`:

```
void CUrl::deleteParam(const string &p) {

/*
** Alle Parameterpaare durchlaufen
*/
  for(parameter_container::iterator iter=m_parameters.begin();
      iter!=m_parameters.end();
      ++iter)

/*
** Wird Parametername gefunden?
** => Parameter-Paar löschen
*/
    if(iter->first==p) {
      m_parameters.erase(iter);
      buildUrl();
      return;
    }
}
```

Listing 361: Die Methode deleteParam von CUrl

setValue

Wir hatten im vorigen Rezept gesehen, dass ein Parameter als bloßer Wert vorkommen kann. Um dieser Tatsache Rechnung zu tragen, ergänzen wir `CUrl` um eine Methode `setValue`, mit der ein solcher Wert zur Parameterliste hinzugefügt werden kann. Sie achtet darauf, dass derselbe Wert nicht mehrmals auftritt:

```
void CUrl::setValue(const string &v) {

/*
** Alle Parameterpaare durchlaufen
*/
  for(parameter_container::iterator iter=m_parameters.begin();
      iter!=m_parameters.end();
      ++iter)
```

Listing 362: Die Methode setValue von CUrl

```
/*
** Wird Wert gefunden?
** => Funktion beenden
*/
    if(iter->second==v)
      return;

/*
** Wert nicht gefunden
** => Wert wird angelegt
*/
  m_parameters.push_back(make_pair("",v));
  buildUrl();
}
```

Listing 362: Die Methode setValue von CUrl (Forts.)

Die Klasse `CUrl` finden Sie auf der CD in den `CUrl`-Dateien.

68 Wie wird eine URL korrekt dekodiert?

Das Dekodieren ist die Umkehrfunktion zu `encodeString` und `encodeUrl` aus Rezept 1.

decodeString

Wie bei der URL-Kodierung wollen wir zuerst eine statische Methode `decodeString` implementieren, die einen beliebigen String dekodiert:

```
string CUrl::decodeString(const string &s) {
  string tmp;
  size_t pos=0;
  while(pos<s.length()) {

/*
** Leerzeichen ?
*/
    if(s[pos]=='+') {
      tmp+=" ";
      pos++;
```

Listing 363: Die Methode decodeString

```
    }

/*
** Normales Zeichen
*/
    else if(s[pos]!='%')
      tmp+=s[pos++];
    else {

/*
** Mit % kodiertes Zeichen
*/
      tmp+=static_cast<char>(hexToValue(s.substr(pos+1,2)));
      pos+=3;
    }
  }
  return(tmp);
}
```

Listing 363: Die Methode decodeString (Forts.)

Betrachtet man die Methode `decodeString` ein wenig genauer, dann fällt auf, dass ein Leerzeichen auch mit + kodiert werden kann. Dieser Sonderfall muss beim Dekodieren berücksichtigt werden.

decodeUrl

Die folgende Methode `decodeUrl` dekodiert die wesentlichen Teile der URL:

```
void CUrl::decodeUrl() {
  for(folder_container::iterator fiter=m_folders.begin();
      fiter!=m_folders.end();
       ++fiter)
    *fiter=decodeString(*fiter);

  for(parameter_container::iterator piter=m_parameters.begin();
      piter!=m_parameters.end();
      ++piter) {
    piter->first=decodeString(piter->first);
    piter->second=decodeString(piter->second);
  }
}
```

Listing 364: Die Methode decodeUrl

getDecodedUrl

Nun fehlt nur noch die Methode `getDecodedUrl`, die eine Kopie der URL anfertigt und diese dann dekodiert zurückliefert:

```
CUrl CUrl::getDecodedUrl() {
  CUrl tmp=*this;
  tmp.decodeUrl();
  tmp.buildUrl();
  return(tmp);
}
```

Listing 365: Die Methode getDecodedUrl

Die Klasse `CUrl` finden Sie auf der CD in den `CUrl`-Dateien.

69 Wie wird eine TCP/IP-Verbindung (Internet) hergestellt?

Die Beantwortung dieser Frage erfordert ein system- und compilerspezifisches Programmieren. Wir werden die plattform- und compilerabhängigen Komponenten jedoch so gering wie möglich halten, um eine einfache Portierung zu ermöglichen.

Als Beispielimplementierung nehmen wir das Betriebssystem Windows und den Visual-C++-Compiler von Microsoft.

Entworfen werden soll ein Klasse, die folgende Eigenschaften besitzt:

Sie besitzt eine statische Methode `initialize`, die einmalig im Programm aufgerufen wird und bestimmte Teile des Systems initialisiert.

Dem Konstruktor werden die Adresse und der Port des Servers übergeben, mit dem die Verbindung aufgenommen werden soll.

Der Destruktor gibt alle Ressourcen wieder frei und trennt eine eventuell noch bestehende Verbindung.

Eine Methode `is_available` gibt Auskunft darüber, ob die gewünschte Verbindung zur Verfügung steht.

Mit `send` können Daten über die erstellte Verbindung gesendet werden. Dabei besitzt die Methode als Parameter die Adresse des Speichers, aus dem gesendet werden soll, und die Anzahl der zu sendenden Daten.

Die Methode `receive` empfängt Daten über die Verbindung. Ihr muss über Funktionsparameter mitgeteilt werden, wie viele Daten gelesen werden und wo sie abgelegt werden sollen.

Die Methoden `send` und `receive` dürfen dabei blockierende Methoden sein. Für `receive` bedeutet das zum Beispiel, dass die Methode erst beendet wird, wenn genau so viele Daten empfangen wurden wie gefordert.

Im Normalfall würde man blockierende Methoden mit Hilfe so genannter Callback-Methoden vermeiden. Da wir uns aber auf die Fahne geschrieben haben, die Implementierung zwecks leichter Portierbarkeit einfach zu halten, werden wir auf sie verzichten.

Es soll hier noch einmal ausdrücklich darauf hingewiesen werden, dass bei einem reinen MFC-Projekt weitaus elegantere Methoden zum Einsatz kämen. Die folgenden Zeilen sollen daher nicht als Paradebeispiel der Windows-Programmierung verstanden werden, sondern als einfache Lösungsmechanismen, die eine Anpassung an die eigenen Bedürfnisse oder andere Plattformen/Compiler ohne großen Aufwand ermöglichen.

CSpecificINetCon

Die Klasse soll `CSpecificINetCon` heißen. Abgesehen vom Attribut `m_available`, das speichert, ob die Verbindung verfügbar ist, sind alle anderen Attribute umgebungsspezifisch.

Klassendefinition

```
class CSpecificINetConn {
private:
  static WSADATA m_wsaData;
  CSocket *m_socket;
  CSocketFile *m_sfile;
  CArchive *m_archive;
  bool m_available;

public:
  bool isAvailable() {
```

Listing 366: Die Klassendefinition von CSpecificINetCon

```
    return(m_available);
  }
  static void initialize();
  CSpecificINetConn(const std::string &host, unsigned int port);
  virtual ~CSpecificINetConn();
  int receive(void *buffer, int bsize);
  int send(const void *buffer, int bsize);
};
```

Listing 366: Die Klassendefinition von CSpecificINetCon (Forts.)

Damit die MFC bei einer Konsolen-Anwendung benutzt werden kann, muss dies bei den Projekt-Eigenschaften eingestellt werden.

Trace-Makros

Um den Aufbau der Internet-Verbindung und später auch die Kommunikation mit den Servern besser mitverfolgen zu können, definieren wir uns zwei Trace-Makros, die eine entsprechende Nachricht auf der Konsole ausgeben.

`CBTRACELN` führt eine Ausgabe mit `endl`, `CBTRACE` eine Ausgabe ohne `endl` durch.

Sind die Ausgaben nicht erwünscht, braucht lediglich die erste Zeile mit `//` auskommentiert zu werden.

```
#define CBTR // Definieren für Trace-Ausgaben
#ifdef CBTR
#include <iostream>
#define CBTRACELN(s) std::cout << (s) << endl;
#define CBTRACE(s) std::cout << (s);
#else
#define CBTRACELN(s)
#define CBTRACE(s)
#endif
```

Listing 367: Die Trace-Makros

initialize

In unserem konkreten Fall einer Implementierung für Windows unter Visual C++ übernimmt die `initialize`-Methode die Aufgaben, MFC und Winsock zu initialisieren (bei Konsolen-Anwendungen eine manuell auszuführende Notwendigkeit).

```
WSADATA CSpecificINetConn::m_wsaData={0,0,0,0,0,0,0,0};

void CSpecificINetConn::initialize() {

/*
** Microsoft Foundation Classes initialisieren
*/
  if (!AfxWinInit(GetModuleHandle(NULL), NULL, GetCommandLine(), 0))
    throw internet_error("CSpecificINetConn::initialise: \
MFC-Initialisierung fehlgeschlagen "+toString(GetLastError()));

/*
** Winsock initialisieren
*/
  if(WSAStartup(MAKEWORD( 2, 0 ), &m_wsaData) != 0)
    throw internet_error("CSpecificINetConn::initialise: \
Winsock 2.0-Initialisierung fehlgeschlagen "+
                         toString(GetLastError()));
}
```

Listing 368: Die Methode initialize

Konstruktor

Der Konstruktor legt die benötigten Objekte an und stellt die Verbindung mit dem Server her. Er ist nach außen hin gegen Ausnahmen abgedichtet. Erfolg oder Misserfolg teilt er über das Attribut `m_available` mit.

```
CSpecificINetConn::CSpecificINetConn(const string &host, unsigned int port)
: m_socket(0), m_sfile(0), m_archive(0), m_available(false) {
  try {

/*
** Socket anlegen und interne Verbindungen herstellen
*/
    m_socket=new CSocket();
    if(m_socket->Create()) {
      CBTRACELN("Socket kreiert!");

/*
** Socketfile und Archiv für den Datenempfang anlegen
```

Listing 369: Der Konstruktor von CSpecificINetCon

```
** und mit dem Socket verbinden
*/
      m_sfile=new CSocketFile(m_socket);
      m_archive=new CArchive(m_sfile,CArchive::load,10000);
      CBTRACELN("CSocketFile und CArchive konstruiert!");

/*
** Verbindung mit Server, dessen Namen in host gespeichert ist,
** über Port port aufnehmen
*/
      if(m_socket->Connect(host.c_str(),port)) {
        CBTRACELN("Socket verbunden mit "+host+
                  " Port "+toString(port));
        m_available=true;
      }
    }
  }
  catch(...) {
    CBTRACELN("Ausnahme in CSpecificINetConn-Konstruktor");
  }
}
```

Listing 369: Der Konstruktor von CSpecificINetCon (Forts.)

Destruktor

Der Destruktor trennt die Verbindung durch Zerstörung der daran beteiligten Objekte:

```
CSpecificINetConn::~CSpecificINetConn() {
  if(m_archive)
    delete(m_archive);
  if(m_sfile)
    delete(m_sfile);
  if(m_socket)
    delete(m_socket);
}
```

Listing 370: Der Destruktor von CSpecificINetCon

receive

Die Methode `receive` liest maximal `bsize` Bytes über die Verbindung in den Puffer `buffer`.

Sie fängt alle in ihr auftretenden Ausnahmen ab. Die Methode gibt die Anzahl der tatsächlich gelesenen Bytes zurück. Sollte ein Fehler auftreten, liefert sie den Wert 0 zurück.

```
int CSpecificINetConn::receive(void *buffer, int bsize){
  int res=0;
  try {
    res=m_archive->Read(buffer,bsize);
  }
  catch(...) {
    CBTRACELN("Ausnahme in CSpecificINetConn::receive");
  }
  return(res);
}
```

Listing 371: Die Methode receive

send

Die Methode `send` sendet `bsize` im Puffer `buffer` befindliche Bytes über die entsprechende Verbindung. Sie liefert die Anzahl der tatsächlich gesendeten Bytes zurück.

Alle Ausnahmen werden innerhalb der Methode aufgefangen, bei einem Fehler wird der Wert 0 zurückgegeben.

```
int CSpecificINetConn::send(const void *buffer, int bsize){

  int res=0;
  try {
    res=m_socket->Send(buffer,bsize);
  }
  catch(...) {
    CBTRACELN("Ausnahme in CSpecificINetConn::send");
  }
  if(res==SOCKET_ERROR) {
    CBTRACELN("SOCKET_ERROR in CSpecificINetConn::send "+
              toString(m_socket->GetLastError()));
```

Listing 372: Die Methode send

```
    return(0);
  }
  return(res);
}
```

Listing 372: Die Methode send (Forts.)

internet_error

Um eine eigene, einheitliche Ausnahme für alle mit dem Internet auftretenden Probleme zu haben, definieren wir `internet_error`:

```
class internet_error : public std::runtime_error {
public:
  explicit internet_error(const std::string &error)
  : runtime_error(error)
  { }
};
```

Listing 373: Die Ausnahmeklasse internet_error

Die Klassen `CSpecificINetCon` und `internet_error` finden Sie auf der CD in den `CSpecificINetCon`-Dateien.

70 Wie können Zeilen über eine TCP/IP-Verbindung (Internet) gesendet und empfangen werden?

Die Lösung dieses Problems hat eher einen formellen Hintergrund, dient doch die hier vorgestellte Klasse `CInternetConnection` als Grundlage aller weiteren Klassen, die Daten über TCP/IP austauschen.

CInternetConnection

Die Klasse `CInternetConnection` erbt die plattform- und compilerspezifische Funktionalität für TCP/IP-Verbindungen von `CSpecificINetCon` und ergänzt sie um plattformunabhängige Methoden[1], die später benötigt werden.

1. Plattformunabhängig in dem Sinne, als dass sie nur auf die Funktionalität von `CSpecificINetCon` zurückgreift und keine systemspezifischen Ressourcen aufruft

Klassendefinition

Die Klasse `CInternetConnection` besitzt keine eigenen Attribute. Sie definiert lediglich die Konstanten `EOL` (Zeilenendekennung `\n`), `CR` (Wagenrücklauf `\r`) und `CRLF` (die Zeichenkombination `\r\n`.)

Darüber hinaus beinhaltet sie Methoden, die das Senden und Empfangen von Zeilen ermöglichen.

Die Klasse besitzt die in Rezept 69 vorgestellten Trace-Makros.

```
class CInternetConnection : public CSpecificINetConn {
public:
  static const char EOL='\n';
  static const char CR='\r';
  static const std::string CRLF;

  CInternetConnection(const std::string &host, unsigned int port);

  std::string receiveLine();
  int sendLine(const std::string &line);
  int sendLineCRLF(const std::string &line);
};

// Initialisierung von CRLF in der cpp-Datei:

const std::string CInternetConnection::CRLF="\r\n";
```

Listing 374: Die Klassendefinition von CInternetConnection

Konstruktor

Der Konstruktor schleift die Funktionalität des `CSpecificINetCon`-Konstruktors durch:

```
CInternetConnection::CInternetConnection(const string &host,
                                         unsigned int port)
: CSpecificINetConn(host,port)
{ }
```

Listing 375: Der Konstruktor von CInternetConnection

receiveLine

Die Methode liest über die bestehende Verbindung eine Zeile (bis zur Endekennung `EOL`) in einen String ein und liefert diesen dann zurück. Die Endezeichen der Zeile (`CR` und `LF`) werden nicht mit in den String aufgenommen.

Sollte keine Verbindung mehr bestehen, liefert die Methode einen leeren String zurück.

```
string CInternetConnection::receiveLine(){
  string line;
  char c;

/*
** Schleife läuft, bis EOL (LF) empfangen wurde
*/
  do {

/*
** Kein Zeichen mehr empfangen?
** => Abbruch und bisher empfangene Zeichen zurückliefern
*/
    if(!receive(&c,1))
      return(line);

/*
** CR und LF (EOL) überlesen
*/
    if((c!=CR)&&(c!=EOL))
      line+=c;
  } while (c!=EOL);

  return(line);
}
```

Listing 376: Die Methode receiveLine

Die Methode muss die Zeichen einzeln lesen, weil auf eine blockierende `read`-Methode zurückgegriffen wird. Es darf daher keinesfalls über die Zeilen-Endekennung hinaus gelesen werden.

sendLine und sendLineCRLF

`sendLine` und `sendLineCRLF` senden jeweils eine Zeile über die bestehende Verbindung, einmal ohne und einmal mit einem abschließenden `CRLF`.

```
int CInternetConnection::sendLine(const std::string &line) {
  return(send(line.c_str(), line.size()));
}

//*********

int CInternetConnection::sendLineCRLF(const std::string &line) {
  return(sendLine(line+CRLF));
}
```

Listing 377: Die Methoden sendLine und sendLineCRLF

Die Klasse `CInternetConnection` finden Sie auf der CD in den `CInternetConnection`-Dateien.

71 Wie kann ein HTTP-Request formuliert werden?

Ein HTTP-Request ist die Voraussetzung für das Anfordern einer Datei über das HTTP-Protokoll, also das Protokoll, auf dem das World Wide Web aufbaut.

Ein HTTP-Request besitzt einen Kopf, dem so genannte Request-Header folgen. Jede Zeile wird mit CR und LF abgeschlossen.

Das Ende des Requests bildet eine leere Zeile, die ebenfalls mit CR und LF abgeschlossen ist. Abbildung 32 stellt den Zusammenhang grafisch dar.

Der Request-Kopf besteht aus drei mit Leerzeichen getrennten Bereichen.

Der erste Teil definiert die Methode, also was der Server machen soll (z.B. ein Dokument senden, ein Dokument speichern, Dokument-Informationen senden etc.)

Im zweiten Rezept wird die angeforderte Ressource definiert.

Der dritte und letzte Teil definiert das Protokoll und die verwendete Version.

Abbildung 33 zeigt den Zusammenhang an einem Beispiel.

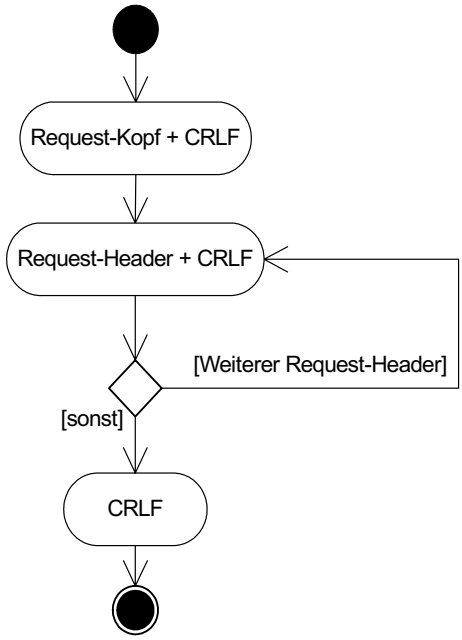


Abbildung 32: Der Aufbau eines HTTP-Requests

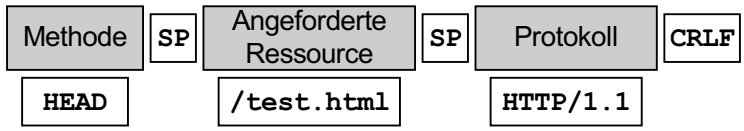


Abbildung 33: Der Kopf des HTTP-Requests

Ein konkretes Beispiel sehen Sie hier:

```
GET /main/main.asp?page=home/default HTTP/1.1
Host: www.addison-wesley.de
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

Diese drei Zeilen fordern die Startseite der Homepage von Addison-Wesley an.

CHttpRequest

Um einen HTTP-Request einfach formulieren zu können, implementieren wir eine Klasse `CHttpRequest`, die uns die wichtigsten Request-Header zur Verfügung stellt.

Klassendefinition

```
class CHttpRequest {
public:
  enum RequestType {rtGET=0, rtPOST, rtHEAD, rtPUT, rtDELETE,
                    rtTRACE, rtCONNECT};
  static const std::string const RTStrings[];
  static const std::string const PTStrings[];
  static const std::string UAExplorer6;
  static const std::string UANetscape6;

private:
  RequestType m_requestType;
  ProtocolType m_protocol;
  std::string m_filePath;
  std::string m_host;
  std::string m_range;
  std::string m_referer;
  std::string m_userAgent;
  std::string m_maxForwards;
};
```

Listing 378: Die Klassendefinition von CHttpRequest

Die Klasse besitzt vier statische Attribute. `RTStrings` ist ein Feld mit den Namen der Request-Methoden im String-Format.

In `PTStrings` sind die String-Versionen der Protokoll-Typen abgelegt.

`UAExplorer6` und `UANetscape6` sind Strings für den User-Agent-Header. Näheres erfahren Sie dort.

```
const std::string CHttpRequest::UAExplorer6=
           "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)";
const std::string CHttpRequest::UANetscape6=
"Mozilla/5.0 (Windows; U; Win98; en-US; m18) Gecko/20010131 \
Netscape6/6.01";

const string const CHttpRequest::RTStrings[]={"GET", "POST", "HEAD",
                                               "PUT", "DELETE",
                                               "TRACE", "CONNECT"};
```

```
const string const CHttpRequest::PTStrings[]={"HTTP/1.0",
                                               "HTTP/1.1"};
```

Konstruktor

Der Konstruktor macht nicht viel, außer das Attribut `m_requestType` mit der gängigsten Request-Methode zu initialisieren und das Protokoll auf HTTP 1.1 zu setzen.

```
  CHttpRequest()
  : m_requestType(rtGET), m_protocol(HTTP11)
  { }
```

Listing 379: Der Konstruktor von CHttpRequest

Methoden für den Request-Kopf

Mit den folgenden drei Methoden können die Komponenten des Request-Kopfes verändert werden.

Bei `setFilePath` muss darauf geachtet werden, dass der Pfad auch den Dateinamen enthält und weder einen führenden noch einen abschließenden / besitzen darf.

```
void setRequestType(RequestType rt) {
  m_requestType=rt;
}

//*********

void setProtocol(ProtocolType pt) {
  m_protocol=pt;
}

//*********

void setFilePath(const std::string &filepath) {
  m_filePath=filepath;
}
```

Listing 380: Die Methoden für den Request-Kopf

Host

Der Request-Header `Host` ist ab HTTP 1.1 Pflicht. Er definiert, mit welchem Host Kontakt aufgenommen werden soll (bzw. mit welchem Host der Client denkt, Kontakt aufzunehmen.)

Von der technischen Seite der Verbindungsaufnahme her ist der Header uninteressant, weil der Verbindungsaufbau auf der Ebene unserer Klasse `CSpecificINetCon` bereits abgeschlossen ist, wenn ein Request-Header versendet wird.

Der Host-Header besitzt die Syntax

```
Host: hostname:port
```

Sollte der Standard-Port (bei HTTP ist dies der Port 80) verwendet werden, dann wird der Port (und der trennende Doppelpunkt) nicht angegeben.

Die Methoden bieten mehrere Möglichkeiten, Host und Port anzugeben.

```
void CHttpRequest::setHost(const string &host, const string &port) {
  m_host=host;
  if((port!="")&&(port!="80"))
    m_host+=":"+port;
}

//*********

void setHost(const std::string &host, unsigned int port) {
  setHost(host,toString(port));
}

//*********

void setHost(const CUrl &url) {
  setHost(url.getDomain(), url.getPort());
}

//*********

void clearHost() {
```

Listing 381: Die Methoden für Host

```
  m_host="";
}
```

Listing 381: Die Methoden für Host (Forts.)

Max-Forwards

`Max-Forwards` wird dem Request im Normalfall nur bei der Request-Methode TRACE hinzugefügt. Der Header definiert, durch wie viele Proxies der TRACE-Request laufen darf, bis er zurückgeschickt wird.

Die einzelnen Stationen tragen sich dabei im Response-Header `Via` ein.

```
void setMaxForwards(const std::string &mf) {
  m_maxForwards=mf;
}

//*********

void setMaxForwards(int mf) {
  setMaxForwards(toString(mf));
}
```

Listing 382: Die Methoden für Max-Forwards

Range

Mit Hilfe des Headers `Range` können vom Server nur bestimme Teile einer Datei angefordert werden. Der Header ist folgendermaßen aufgebaut:

```
Range: bytes=anf-end
```

Dabei definieren `anf` und `end` den Anfang und das Ende des gewünschten Teils und werden durch konkrete Zahlenwerte ersetzt.

Dieser Header ist wichtig, um eine Resume-Möglichkeit für abgebrochene Übertragungen zu realisieren.

```
void CHttpRequest::setRange(unsigned long begin, unsigned long end) {
  m_range="bytes="+toString(begin)+"-"+toString(end);
}

//*********

void CHttpRequest::setRange(CHugeNumber begin, CHugeNumber end) {
  m_range="bytes="+begin.getAsString()+"-"+end.getAsString();
}

//*********

void clearRange() {
  m_range="";
}
```

Listing 383: Die Methoden für Range

Referer

Der Header `Referer` definiert, über welche Ressource die gerade angeforderte Ressource erreicht wurde. Er sollte immer mit angegeben werden.

Würde beispielsweise auf der Seite *http://www.pearsoneducation.de/* der Link zu Addison-Wesley angeklickt, dann würde der Client im GET-Request für die Addison-Wesley-Seite die URL *http://www.pearsoneducation.de/* als Referer eintragen.

Werden zu einem HTML-Dokument Bilder nachgeladen, dann haben die GET-Requests der Bilder das HTML-Dokument als Referer.

Auf diese Weise kann der Server feststellen, ob eine Grafik von einem auf dem eigenen Server befindlichen Dokument oder einem fremden Dokument geladen wurde. Manche Server (z.B. der von Microsoft) verbietet fremden Dokumenten das Laden von Bildern.

```
void setReferer(const std::string &ref) {
  m_referer=ref;
}

//*********
```

Listing 384: Die Methoden für Referer

```
void setReferer(const CUrl &ref) {
  m_referer=ref.getUrl();
}
```

Listing 384: Die Methoden für Referer (Forts.)

User-Agent

Über den Header `User-Agent` kann das Client-Programm dem Server seine Identität mitteilen. Manchmal macht es Sinn, sich über diesen Header als ein bekannter Browser auszugeben, weil manche Server auf unbekannte Clients misstrauisch reagieren.

Deswegen besitzt die Klasse die vorgefertigten Strings der beiden bekanntesten Browser.

```
void setUserAgent(const std::string &agent) {
  m_userAgent=agent;
}
```

Listing 385: Die Methode für User-Agent

setFromUrl

Um den Vorgang der Formulierung eines Requests etwas zu beschleunigen, definieren wir die Methode `setFromUrl`, die aus einem `CUrl`-Objekt (Rezept 64) alle für den Request notwendigen Informationen herauszieht.

Darüber hinaus werden andere notwendige Header mit gängigen Standardwerten versehen. Individuelle Änderungen können danach immer noch mit den entsprechenden `set`-Methoden vorgenommen werden.

```
void CHttpRequest::setFromUrl(const CUrl &url) {
  m_requestType=rtGET;
  m_protocol=HTTP11;
  m_userAgent=UAExplorer6;
  setHost(url);

/*
** Filepath mit Parametern erzeugen
*/
  m_filePath=url.getPath();
```

Listing 386: Die Methode setFromUrl von CHttpRequest

```
  string params=url.getParametersAsString();
  if(params!="")
    m_filePath+="?"+params;
}
```

Listing 386: Die Methode setFromUrl von CHttpRequest (Forts.)

```
string CHttpRequest::getRequest() const{
  string request;

/*
** GET- oder HEAD-Request?
*/
  switch(m_requestType) {
    case rtGET:
    case rtPOST:
    case rtHEAD:

/*
** Alle definierten Header zum Request hinzufügen
*/
      request+=RTStrings[m_requestType]+" /"+
               m_filePath+" "+
               PTStrings[m_protocol]+CInternetConnection::CRLF;
      if(m_host!="")
        request+="Host: "+m_host+CInternetConnection::CRLF;
      if(m_range!="")
        request+="Range: "+m_range+CInternetConnection::CRLF;
      if(m_referer!="")
        request+="Referer: "+m_referer+CInternetConnection::CRLF;
      if(m_userAgent!="")
        request+="User-Agent: "+m_userAgent+
                 CInternetConnection::CRLF;
      request+=CInternetConnection::CRLF;
      break;

/*
** TRACE-Request?
*/
    case rtTRACE:
```

Listing 387: Die Methode getRequest

```
/*
** Alle definierten Header zum Request hinzufügen
*/
      request+=RTStrings[m_requestType]+" /"+
               m_filePath+" "+
               PTStrings[m_protocol]+CInternetConnection::CRLF;
      if(m_host!="")
        request+="Host: "+m_host+CInternetConnection::CRLF;
      if(m_maxForwards!="")
        request+="Max-Forwards: "+m_maxForwards+
                 CInternetConnection::CRLF;
      if(m_range!="")
        request+="Range: "+m_range+CInternetConnection::CRLF;
      if(m_referer!="")
        request+="Referer: "+m_referer+CInternetConnection::CRLF;
      if(m_userAgent!="")
        request+="User-Agent: "+m_userAgent+
                 CInternetConnection::CRLF;
      request+=CInternetConnection::CRLF;
      break;

    default:
      return("");
  }

  return(request);
}
```

Listing 387: Die Methode getRequest (Forts.)

Die Klasse `CHttpRequest` finden Sie auf der CD in den `CHttpRequest`-Dateien.

72 Wie kann die Antwort eines HTTP-Servers ausgewertet werden?

Nachdem wie im vorigen Rezept beschrieben ein HTTP-Request abgesetzt wurde, schickt der Server eine Antwort, einen so genannten HTTP-Response, zurück.

Dieser kann je nach Request-Methode aus reinen Informationen oder aus Informationen und angeforderten Daten bestehen.

Abbildung 35 zeigt den Aufbau eines solchen HTTP-Response.

In diesem Rezept ist für uns nur der obere, mit Server-Response beschriftete Block interessant. Betrachten wir den Kopf der Server-Response einmal genauer, denn mit ihm teilt der Server den Status des Requests mit. Der Kopf ist in Abbildung 34 dargestellt.

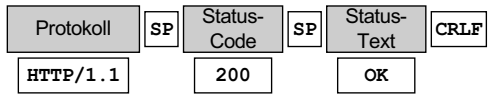


Abbildung 34: Der Kopf der Server-Response

CHttpResponse

Um die Informationen einer Server-Anwort auswerten zu können, definieren wir eine Klasse `CHttpResponse`, die alle für uns relevanten Informationen extrahiert und zugänglich macht.

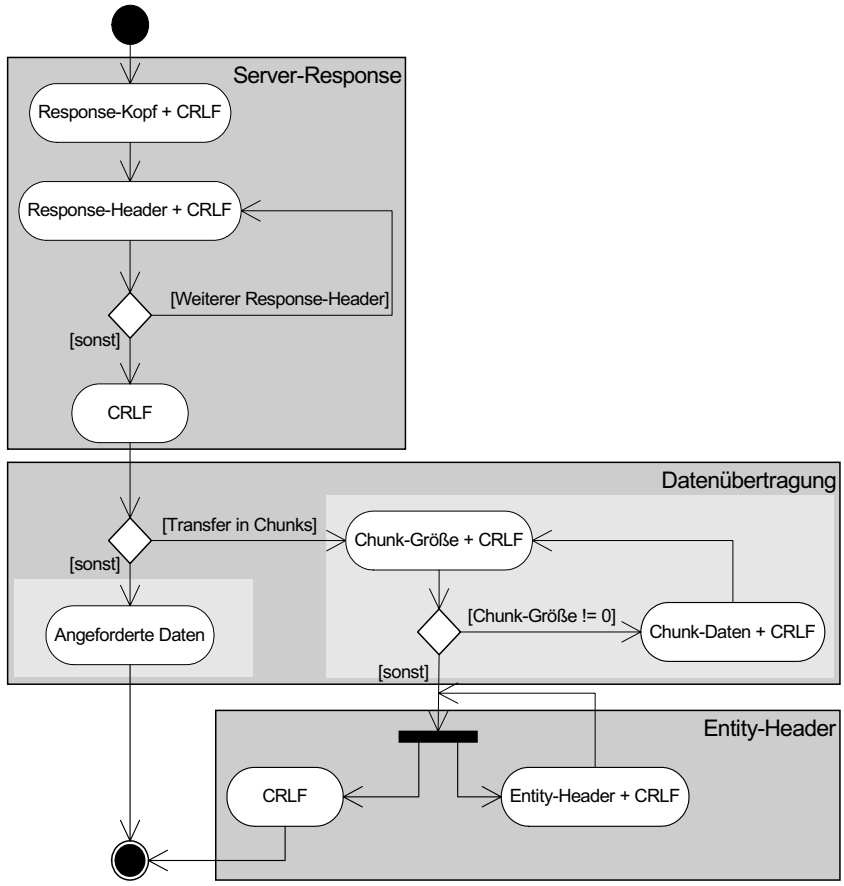


Abbildung 35: Die Antwort eines HTTP-Servers

Klassendefinition

```
class CHttpResponse {
public:

private:
  std::string m_acceptRanges;
  std::string m_eTag;
  std::string m_location;
  std::string m_retryAfter;
  std::string m_server;
  std::string m_contentLanguage;
  std::string m_contentLength;
  std::string m_contentLocation;
  std::string m_contentType;
  std::string m_transferEncoding;
  std::string m_setCookie;
  std::string m_connection;
  std::string m_via;
  CMoment m_date;
  CMoment m_expires;
  CMoment m_lastModified;
  std::string m_serverStatus;
  std::string m_protocol;
  int m_serverStatusCode;
  std::vector<std::string> m_unknownHeaders;
};
```

Listing 388: Die Klassendefinition von CHttpResponse

Konstruktor

Der Konstruktor setzt den Status-Code auf 0, was für unsere Klasse als Vereinbarung gilt, dass ein Objekt noch keinen Response enthält.

```
CHttpResponse()
: m_serverStatusCode(0)
{ }
```

Listing 389: Der Konstruktor von CHttpResponse

Methoden für den Response-Kopf

Um an die drei Komponenten des Response-Kopfes heranzukommen, existieren die folgenden Methoden:

```
std::string getRanges() const {
  return(m_acceptRanges);
}

//*********

std::string getProtocol() const {
  return(m_protocol);
}

//*********

int getStatusCode() const {
  return(m_serverStatusCode);
}
```

Listing 390: Die Methoden für den Response-Kopf

Accept-Ranges

Dieser Header definiert, ob der Server in der Lage ist, nur Teile einer angeforderten Ressource zu senden. Ist der Header vorhanden, ist entweder der Wert `none` oder `bytes` angegeben.

Die Angabe dieses Headers ist keine Pflicht. Deswegen kommt es häufig vor, dass Server Teile einer Datei schicken können, obwohl dieser Header nicht in der Response enthalten war.

```
std::string getRanges() const {
  return(m_acceptRanges);
}
```

Listing 391: Die Methode getRanges

Connection

Der Header `Connection` teilt dem Client mit, wie der Server mit der bestehenden Verbindung nach der Übertragung verfährt.

Der Wert `close` besagt, dass die Verbindung nach der Übertragung abgebrochen wird.

Mit `keep-alive` teilt der Server mit, dass er die Verbindung nach der Übertragung aufrecht erhält.

Ist dieser Header nicht definiert, verwendet HTTP/1.0 standardmäßig `close` und HTTP/1.1 `keep-alive`.

```
std::string getConnection() const {
  return(m_connection);
}
```

Listing 392: Die Methode getConnection

Content-Language

Der Header `Content-Language` zählt zu den so genannten Entity-Headern, gibt also Auskunft über die angeforderte Ressource.

Dieser Header speziell teilt mit, in welcher Sprache die Ressource verfasst ist. Die Sprache wird als die im Internet übliche zweistellige Abkürzung angegeben (z. B. de für Deutschland.)

Ist eine Ressource für mehrere Sprachen gedacht (z. B. ein Programm mit Sprachwahl für die Menüs), dann sind die einzelnen Angaben mit Komma getrennt.

```
std::string getcontentLanguage() const {
  return(m_contentLanguage);
}
```

Listing 393: Die Methode getContentLanguage

Content-Length

Der Entity-Header `Content-Length` liefert die Größe der angeforderten Ressource in Bytes.

Dieser Header ist wichtig für eine eventuelle Resume-Funktion, denn ist die komplette Länge der Ressource nicht bekannt, kann auch nicht bestimmt werden, ob die Ressource bereits komplett geladen wurde oder nicht.

```
std::string getContentLength() const {
  return(m_contentLength);
}
```

Listing 394: Die Methode getContentLength

Content-Location

Über den Entity-Header `Content-Location` kann der Server mitteilen, an welcher Stelle die angeforderte Ressource liegt. Üblicherweise wird diese Information in Form einer relativen oder absoluten URL zur Verfügung gestellt.

```
std::string getContentLocation() const {
  return(m_contentLocation);
}
```

Listing 395: Die Methode getContentLocation

Content-Type

Dieser Entity-Header liefert den MIME-Type der angeforderten Ressource. Bei einem HTML-Dokument wäre dies beispielsweise `text/html`.

```
std::string getContentType() const {
  return(m_contentType);
}
```

Listing 396: Die Methode getContentType

Date

Mit dem Header `Date` teilt der Server das aktuelle Datum und die Uhrzeit mit. Das Format ist aufgebaut wie in RFC1123 definiert:

```
Fri, 30 May 2003 16:50:11 GMT
```

Die Klasse `CMoment` stellt eine Methode `convertGmtToMet` zur Verfügung, mit der die angegebene Zeit in die für Deutschland gültige Zeitzone umgewandelt werden kann.

```
CMoment getDate() const {
  return(m_date);
}
```

Listing 397: Die Methode getDate

ETag

Das ETag ist eine eindeutige Bezeichnung für die angeforderte Ressource auf dem Server. Sie kann für Verwaltungszwecke eingesetzt werden.

```
std::string getETag() const {
  return(m_eTag);
}
```

Listing 398: Die Methode getETag

Expires

Dieser Entitäts-Header definiert, wann sich die angeforderte Ressource ändern kann oder ungültig wird. Es ist aber nicht zwingend, dass zu dem in `Expires` angegebenen Zeitpunkt tatsächlich eine Änderung der Ressource eintritt bzw. durchgeführt wird.

Der Header verwendet für das Datum das bei `Date` beschriebene Format.

```
CMoment getExpires() const {
  return(m_expires);
}
```

Listing 399: Die Methode getExpires

Last-Modified

Dieser Entitäts-Header teilt das Datum der letzten an der Ressource vorgenommenen Änderungen mit.

Der Header verwendet für das Datum das bei `Date` beschriebene Format.

```
CMoment getLastModified() const {
  return(m_lastModified);
}
```

Listing 400: Die Methode getLastModified

Location

Dieser Header liefert die tatsächliche Position der angeforderten Ressource. Tritt hauptsächlich mit den Statusmeldungen im 300er-Bereich auf.

```
std::string getLocation() const {
  return(m_location);
}
```

Listing 401: Die Methode getLocation

Retry-After

Sollte der Server die Statusmeldung »503 Service Unavailable« liefern, dann kann er mit Retry-After definieren, ab wann der Request voraussichtlich wieder erfolgreich sein wird.

Der Header liefert entweder ein Datum wie bei Date beschrieben oder die Anzahl der zu wartenden Sekunden, bis der Request neu gestellt werden kann.

```
std::string getRetryAfter() const {
  return(m_retryAfter);
}
```

Listing 402: Die Methode getRetryAfter

Server

Dieser Header liefert den Namen des Servers und dessen Version.

```
std::string getServer() const {
  return(m_server);
}
```

Listing 403: Die Methode getServer

Set-Cookie

Definiert ein vom Client zu speicherndes Cookie

```
std::string getCookie() const {
  return(m_setCookie);
}
```

Listing 404: Die Methode getCookie

Transfer-Encoding

Sollte die angeforderte Ressource nicht an einem Stück, sondern in mehreren Teilen (Chunks) gesendet werden, so wird dieser Header mit dem Wert `chunked` gesendet.

```
std::string getTransferEncoding() const {
  return(m_transferEncoding);
}
```

Listing 405: Die Methode getTransferEncoding

Via

Der Header `Via` tritt hauptsächlich im Zusammenhang mit dem TRACE-Request auf. Die einzelnen Proxies und Server tragen sich mit ihrer Adresse und der verwendeten Protokoll-Version in diesem Header ein.

```
std::string getVia() const {
  return(m_via);
}
```

Listing 406: Die Methode getVia

getUnknownHeaders

Alle von der Klasse `CHttpResponse` nicht erkannten Header werden im Vektor `m_unknownHeaders` gespeichert, auf den uns die Methode `getUnknownHeaders` Zugriff gewährt.

```
const std::vector<std::string> &getUnknownHeaders() const {
  return(m_unknownHeaders);
}
```

Listing 407: Die Methode getUnknownHeaders

Hilfsmethoden

Um mit den Informationen der Server-Response besser arbeiten zu können, definieren wir einige Methoden, die bestimmte Sachverhalte für uns in Erfahrung bringen.

Die Methode `isResponseAvailable` liefert beispielsweise einen booleschen Wert zurück, der Auskunft darüber gibt, ob das entsprechende `CHttpResponse`-Objekt einen gültigen Response beinhaltet.

```
bool isResponseAvailable() const {
  return(m_serverStatusCode!=0);
}
```

Listing 408: Die Methode isResponseAvailable

Über die Methode `isContentLengthAvailable` lässt sich in Erfahrung bringen, ob die Daten der angeforderten Ressource geladen werden müssen, bis der Server die Verbindung trennt, oder ob eine exakte Größenangabe vorliegt.

```
bool isContentLengthAvailable() const {
  return(m_contentLength.size()!=0);
}
```

Listing 409: Die Methode isContentLengthAvailable

Mit Hilfe von `isTransferChunked` lässt sich bestimmen, ob die Ressource in mehreren Teilen oder im Ganzen übertragen wird.

```
bool isTransferChunked() const {
  return(insensitive_string("chunked")==m_transferEncoding.c_str());
}
```

Listing 410: Die Methode isTransferChunked

extract

Die Methode `extract` schließlich ist das Herzstück der Klasse. Sie bekommt die einzelnen Zeilen der Server-Response als Vektor übergeben und extrahiert daraus die ihr bekannten Header.

Der zweite Parameter `withHead`, der standardmäßig auf `true` steht, definiert, ob der übergebene Response einen Response-Kopf besitzt oder nicht.

Bei einem Transfer in Chunks können vom Server noch einzelne Entity-Header gesendet werden, die keinen Response-Kopf besitzen. Die `extract`-Methode muss deswegen auch in der Lage sein, Server-Antworten ohne Kopf zu verstehen.

```
void CHttpResponse::extract(const vector<string> &response,
                            bool withHead) {
  CBTRACELN("\nCHttpResponse::extract...\n---------------------");

  vector<insensitive_string> components;
  vector<string>::const_iterator iter=response.begin();

/*
** Ist ein Response-Kopf auszuwerten?
*/
  if(withHead) {
    components=chopIntoWords(insensitive_string(iter->c_str())," ");

/*
** Protokoll
*/
    m_protocol=components[0].c_str();

/*
** Status-Code
*/
    m_serverStatusCode=atoi(components[1].c_str());

/*
** Status-Text
*/
    m_serverStatus=iter->substr(components[0].size()+
                                components[1].size()+2);
    CBTRACELN("\""+m_protocol+"\" \""+toString(m_serverStatusCode)+
              "\" \""+m_serverStatus+"\"");

    m_unknownHeaders.clear();
    ++iter;
  }

  while(iter!=response.end()) {
```

Listing 411: Die Methode extract von CHttpResponse

```
/*
** Header in einzelne Teile zerlegen
*/
    components=chopIntoWords(insensitive_string(iter->c_str())," ");

/*
** Date?
*/
    if(components[0]=="date:") {
      if(components.size()==7) {
        m_date=CMoment(iter->substr(iter->find(":")+2));
        CBTRACELN("Erkannt: Date "+m_date.getDDMMYYYYHHMMSS());
      }
      else {
        CBTRACELN("HEADER FEHLERHAFT: "+ *iter);
      }
    }

/*
** Last-Modified?
*/
    else if(components[0]=="last-modified:") {
      if(components.size()==7) {
        m_lastModified=CMoment(iter->substr(iter->find(":")+2));
        CBTRACELN("Erkannt: Last-Modified "+
                  m_lastModified.getDDMMYYYYHHMMSS());
      }
      else {
        CBTRACELN("HEADER FEHLERHAFT: "+ *iter);
      }
    }

/*
** Expires?
*/
    else if(components[0]=="expires:") {
      if(components.size()==7) {
        m_expires=CMoment(iter->substr(iter->find(":")+2));
        CBTRACELN("Erkannt: Expires "+
                   m_expires.getDDMMYYYYHHMMSS());
      }
```

Listing 411: Die Methode extract von CHttpResponse (Forts.)

```
      else {
        CBTRACELN("HEADER FEHLERHAFT: "+ *iter);
      }
    }

/*
** Accept-Ranges?
*/
    else if(components[0]=="accept-ranges:") {
      m_acceptRanges=components[1].c_str();
      CBTRACELN("Erkannt: Accept-Ranges "+m_acceptRanges);
    }

/*
** ETag?
*/
    else if(components[0]=="etag:") {
      m_eTag=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: ETag "+m_eTag);
    }

/*
** Location?
*/
    else if(components[0]=="location:") {
      m_location=components[1].c_str();
      CBTRACELN("Erkannt: Location "+m_location);
    }

/*
** Server?
*/
    else if(components[0]=="server:") {
      m_server=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Server "+m_server);
    }

/*
** Content-Language?
*/
    else if(components[0]=="content-language:") {
```

Listing 411: Die Methode extract von CHttpResponse (Forts.)

```
      m_contentLanguage=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Content-Language "+m_contentLanguage);
    }

/*
** Content-Length?
*/
    else if(components[0]=="content-length:") {
      m_contentLength=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Content-Length "+m_contentLength);
    }

/*
** Content-Location?
*/
    else if(components[0]=="content-location:") {
      m_contentLocation=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Content-Location "+m_contentLocation);
    }

/*
** Content-Type?
*/
    else if(components[0]=="content-type:") {
      m_contentType=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Content-Type "+m_contentType);
    }

/*
** Connection?
*/
    else if(components[0]=="connection:") {
      m_connection=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Connection "+m_connection);
    }

/*
** Transfer-Encoding?
*/
    else if(components[0]=="transfer-encoding:") {
      m_transferEncoding=iter->substr(iter->find(":")+2);
```

Listing 411: Die Methode extract von CHttpResponse (Forts.)

```
      CBTRACELN("Erkannt: Transfer-Encoding "+m_transferEncoding);
    }

/*
** Set-Cookie?
*/
    else if(components[0]=="set-cookie:") {
      m_setCookie=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Set-Cookie "+m_setCookie);
    }

/*
** Via:
*/
    else if(components[0]=="via:") {
      m_via=iter->substr(iter->find(":")+2);
      CBTRACELN("Erkannt: Via "+m_via);
    }

/*
** Von CHttpResponse nicht unterstützter Header!
** Wird in m_unknownHeaders abgelegt
*/
    else {
      CBTRACELN("UNKNOWN: "+ *iter);
      m_unknownHeaders.push_back(*iter);
    }
    ++iter;
  }
}
```

Listing 411: Die Methode extract von CHttpResponse (Forts.)

Die Klasse `CHttpResponse` finden Sie auf der CD in den `CHttpResponse`-Dateien.

73 Wie kann eine Datei von einem HTTP-Server übertragen werden?

Wir schreiben dazu eine Klasse `CHttpConnection`, die eine Verbindung zu einem HTTP-Server aufnimmt, einen Request absetzt, den Server-Response auswertet und die mitgeschickte Ressource als Datei abspeichert.

CHttpConnection

Die tatsächliche Verbindung ins Internet wird über die Klasse `CInternetConnection` aus Rezept 69 abgewickelt.

Die Klasse macht von den ebenfalls in Rezept 69 vorgestellten Trace-Makros Gebrauch.

Klassendefinition

Als Attribute besitzt die Klasse einen Zeiger auf das für die TCP/IP-Verbindung zuständige `CInternetConnection`-Objekt (`m_iconn`), einen String, der den zuletzt aufgetretenen Fehler enthält (`m_lastError`) und die letzte empfangene Server-Response (`m_lastResponse`.)

```
class CHttpConnection {
private:
  CInternetConnection *m_iconn;
  std::string m_lastError;
  CHttpResponse m_lastResponse;
};
```

Listing 412: Die Klassendefinition von CHttpConnection

Konstruktor

Der simple Konstruktor setzt `m_iconn` auf 0, damit im weiteren Verlauf überprüft werden kann, ob `m_iconn` auf ein gültiges `CInternetConnection`-Objekt zeigt.

```
CHttpConnection::CHttpConnection()
: m_iconn(0)
{ }
```

Listing 413: Der Konstruktor von ChttpConnection

Destruktor

Der Destruktor trennt mit einem Aufruf von `disconnect` die Verbindung und löscht das `CInternetConnection`-Objekt.

```
~CHttpConnection() {
  disconnect();
}
```

Listing 414: Der Destruktor von CHttpConnection

Zugriffs-Methoden

Damit die Attribute `m_lastError` und `m_lastResponse` ansprechbar sind, werden passende Zugriffs-Methoden definiert:

```
std::string getLastError() const {
  return(m_lastError);
}

//*********

const CHttpResponse &getLastResponse() const {
  return(m_lastResponse);
}
```

Listing 415: Die Zugriffsmethoden von CHttpConnection

connect

Die `connect`-Methode stellt über das `CInternetConnection`-Objekt `m_iconn` die Verbindung zum angegebenen Host über den entsprechenden Port her.

```
bool CHttpConnection::connect(const string &host,
                              unsigned int port) {
  CBTRACELN("CHttpConnection::connect: Verbindung wird aufgebaut");

/*
** Eventuell bestehende Verbindung abbrechen
*/
  disconnect();

/*
** Internet-Verbindung mit entsprechender
** Adresse und Port herstellen
*/
  m_iconn=new CInternetConnection(host,port);
  if(m_iconn->isAvailable()) {
    CBTRACELN("Erfolg!");
    return(true);
  }
```

Listing 416: Die connect-Methode für Host und Port

```
/*
** Verbindungsaufbau fehlgeschlagen?
** => CInternetConnection-Objekt löschen
*/
  else {
    m_lastError="Verbindungsaufbau fehlgeschlagen!";
    CBTRACELN("Verbindungsaufbau fehlgeschlagen!");
    delete(m_iconn);
    m_iconn=0;
    return(false);
  }
}
```

Listing 416: Die connect-Methode für Host und Port (Forts.)

Damit die Methode `connect` vielseitiger einsetzbar ist, wird sie für `CUrl`-Objekte überladen:

```
bool CHttpConnection::connect(const CUrl &url) {

/*
** Host und Port aus CUrl holen
*/
  string host=url.getDomain();
  string sport=url.getPort();
  unsigned int port;

  if(sport=="")
    port=80;
  else
    port=atol(sport.c_str());

/*
** connect für Host und Port aufrufen
*/
  return(connect(host,port));
}
```

Listing 417: Die Methode connect für CUrl-Objekte

disconnect

Über `disconnect` wird die Verbindung abgebrochen, indem das `CInternetConnection`-Objekt gelöscht wird.

```
void CHttpConnection::disconnect() {
  if(m_iconn) {
    delete(m_iconn);
    m_iconn=0;
  }
}
```

Listing 418: Die Methode disconnect von CHttpConnection

sendRequest

Die Methode `sendRequest` erzeugt mit einem `CHttpRequest`-Objekt einen Server-Request und schickt diesen an den Server.

```
bool CHttpConnection::sendRequest(const CHttpRequest &request) {
  CBTRACELN("\nsendRequest\n--------------------------------");

/*
** Request senden
*/
  string sreq=request.getRequest();
  CBTRACE(sreq);
  unsigned int sent=m_iconn->sendLine(sreq);

/*
** Sind weniger Zeichen gesendet worden, als
** der Request lang ist?
** => Fehler!
*/
  if(sent<sreq.size()) {
    m_lastError="sendRequest fehlgeschlagen!";
    CBTRACELN("sendRequest fehlgeschlagen!");
    return(false);
  }
  else {
    CBTRACELN("Erfolg!");
```

Listing 419: Die Methode sendRequest von CHttpConnection

```
    return(true);
  }
}
```

Listing 419: Die Methode sendRequest von CHttpConnection (Forts.)

receiveResponse

`receiveResponse` liest die Antwort des Servers zeilenweise ein und speichert sie im übergebenen Vektor.

```
bool CHttpConnection::receiveResponse(vector<string> &response) {
  CBTRACELN("\nreceiveResponse\n------------------------------");
  string line;

/*
** Server-Response zeilenweise einlesen
*/
  while((line=m_iconn->receiveLine())!="") {
    CBTRACELN("Empfangen: "+line);
    response.push_back(line);
  }
  if(response.size()==0)
    return(false);
  else
    return(true);
}
```

Listing 420: Die Methode receiveResponse von ChttpConnection

getFile

Die Methode `getFile` lädt eine Datei von einem HTTP-Server und speichert sie auf einem lokalen Datenträger.

Die Methode bekommt die zu ladende Datei wie im Internet üblich als URL (in unserem Fall als `CUrl`-Objekt) angegeben. Der Dateiname der zu speichernden Datei mit komplettem Pfad wird als zweiter Parameter übergeben.

Der dritte Parameter definiert den an den Server zu schickenden Request.

```
bool CHttpConnection::getFile(const CUrl &url,
                              const string &filename,
                              const CHttpRequest &request) {
  CBTRACELN("\ngetFile\n-----------------------------------");

/*
** Verbindung zum Server herstellen
*/
  if(!connect(url)) {
    m_lastError="Fehler bei connect";
    CBTRACELN("Fehler bei connect");
    return(false);
  }

/*
** Request senden
*/
  if(!sendRequest(request)) {
    m_lastError="Fehler bei sendRequest";
    CBTRACELN("Fehler bei sendRequest");
    return(false);
  }

/*
** Server-Response empfangen
*/
  vector<string> vresponse;
  if(!receiveResponse(vresponse)) {
    m_lastError="Fehler bei receiveResponse";
    CBTRACELN("Fehler bei receiveResponse");
    return(false);
  }

/*
** Response in ein CHttpResponse-Objekt umwandeln
*/
  bool success;
  CHttpResponse response;
  response.extract(vresponse);
  int code=response.getStatusCode();
```

Listing 421: Die Methode getFile von CHttpConnection

```
/*
** Status zwischen 301 und 307?
** => Ressource wurde verschoben. Neue Location aus
** Response holen und getFile neu aufrufen
*/
  if((code>=301)&&(code<=307)&&(response.getLocation()!="")) {
    CBTRACELN("\nRessource verschoben. Neue Location \
bearbeiten...");
    disconnect();
    return(getFile(CUrl(response.getLocation()), filename));
  }

/*
** Alle Status-Codes außerhalb des 200er-Bereichs
** als Fehler betrachten
*/
  else if((response.getStatusCode()<200)||
          (response.getStatusCode()>=300)) {
    m_lastError=toString(response.getStatusCode())+" "+
                          response.getStatusText();
    CBTRACELN("Server meldet Fehler");
    return(false);
  }
  else {

/*
** Je nachdem, ob Transfer an einem Stück oder
** in Chunks benötigt wird, die entsprechende
** Methode aufrufen
*/
    if(!response.isTransferChunked())
      success=normalFileTransfer(filename, response);
    else {
      success=chunkedFileTransfer(filename, response);

/*
** Hat ein Transfer in Chunks stattgefunden?
** => Hinter den Chunks nach eventuellen Entity-Headern
** Ausschau halten
*/
      if(success) {
```

Listing 421: Die Methode getFile von CHttpConnection (Forts.)

```
        vresponse.clear();
        receiveResponse(vresponse);
        response.extract(vresponse,false);
      }
    }
  }

/*
** Komplette Server-Antwort im Objekt speichern
*/
  m_lastResponse=response;
  return(success);
}
```

Listing 421: Die Methode getFile von CHttpConnection (Forts.)

Damit die `getFile`-Methode leichter zu benutzen ist, wird sie mit einer Variante überladen, die einen »handelsüblichen« Request-Header erzeugt und dann die vorige `getFile`-Methode aufruft:

```
bool CHttpConnection::getFile(const CUrl &url,
                              const string &filename) {
  CHttpRequest request;
  request.setFromUrl(url);
  request.setUserAgent(CHttpRequest::UAExplorer6);

  return(getFile(url,filename,request));
}
```

Listing 422: Die Methode getFile ohne Request-Angabe

normalFileTransfer

Die Methode `normalFileTransfer` übernimmt den Datentransfer an einem Stück.

Zunächst wird geprüft, ob in der Server-Antwort eine Information über die genaue Länge der angeforderten Ressource enthalten ist.

Sollte dies der Fall sein, dann werden nur so viele Daten übertragen wie notwendig, weil der Server höchstwahrscheinlich nach Beendigung der Übertragung die Verbindung nicht trennt und ein weiteres Lesen ein Blockieren der `read`-Methode zur Folge hätte.

Ist keine Angabe über die Größe der Ressource vorhanden, dann wird der Server die Verbindung nach der Übertragung der Daten beenden, weil dies die einzige Möglichkeit ist mitzuteilen, dass die Übertragung der Ressource vollständig ist.

Damit auch Dateigrößen über den Wertebereich einer 32-Bit-Variablen hinaus verarbeitet werden können, greifen wir auf das eigene Zahlenformat `CHugeNumber` aus Rezept 119 zurück.

```
bool CHttpConnection::normalFileTransfer(const string &filename,
                                  const CHttpResponse &response) {

/*
** Datei auf lokalem Datenträger zum Schreiben öffnen
*/
  CFile file;
  if(!file.open(filename,true,true)) {
    m_lastError="FEHLER! Lokale Datei konnte nicht zum \
Schreiben geoeffnet werden!";
    CBTRACELN("FEHLER! Lokale Datei konnte nicht zum \
Schreiben geoeffnet werden!");
    return(false);
  }
  CMemory<char> buf(10000);
  unsigned int amount;
  CHugeNumber totallyRead, bufsize("10000");

/*
** Wenn Content-Length verfügbar, dann werden abgezählte
** Daten empfangen
*/
  if(response.isContenLengthAvailable()) {
    CHugeNumber length=response.getContentLength();
    CBTRACELN("Abgezaehlte Daten ("+length.getAsPointedNumber()+
                                  ") werden gelesen!");
    do {

/*
** Darauf achten, dass nicht mehr Daten empfangen
** werden sollen, als benötigt werden (weil read sonst
** blockiert.)
*/
      if(length>=bufsize)
```

Listing 423: Die Methode normalFileTransfer

```
        amount=m_iconn->receive(buf,10000);
      else
        amount=m_iconn->receive(buf,length.getAsUnsignedLong());

      file.writeBlock(buf,amount);
      totallyRead+=static_cast<unsigned long>(amount);
      length-=static_cast<unsigned long>(amount);
      CBTRACE("Bisher uebertragen: "+
              totallyRead.getAsPointedNumber()+" Bytes\r");

/*
** Keine Daten mehr empfangen, obwohl noch nicht alle Daten
** angekommen sind?
** => Fehler!
*/
      if((totallyRead==0)&&(length!=0)) {
        m_lastError="chunkedTransfer: Übertragung abgebrochen";
        CBTRACE("chunkedTransfer: Übertragung abgebrochen");
        return(false);
      }
    } while(length!=0);
  }

/*
** Wenn keine Content-Length verfügbar, dann Daten
** empfangen, bis der Server die Verbindung trennt
*/
  else {
    CBTRACELN("Daten lesen bis zum Ende!");
    do {
      amount=m_iconn->receive(buf,10000);
      file.writeBlock(buf,amount);
      totallyRead+=static_cast<unsigned long>(amount);
      CBTRACE("Bisher uebertragen: "+
              totallyRead.getAsPointedNumber()+" Bytes\r");
    } while(amount);
  }

  CBTRACELN("\nErfolg!");
  return(true);
}
```

Listing 423: Die Methode normalFileTransfer (Forts.)

chunkedFileTransfer

Um die Übertragung einer Ressource in Chunks besser nachvollziehen zu können, werfen wir einen Blick auf Abbildung 36.

Zuerst wird die Größe des folgenden Chunks gesendet. Sollte diese Größe 0 sein, dann ist die Übertragung abgeschlossen.

Bei einer Größe ungleich 0 wird der Chunk empfangen und in die Datei gespeichert. Dann wird die Größe des nächsten Chunks eingelesen usw.

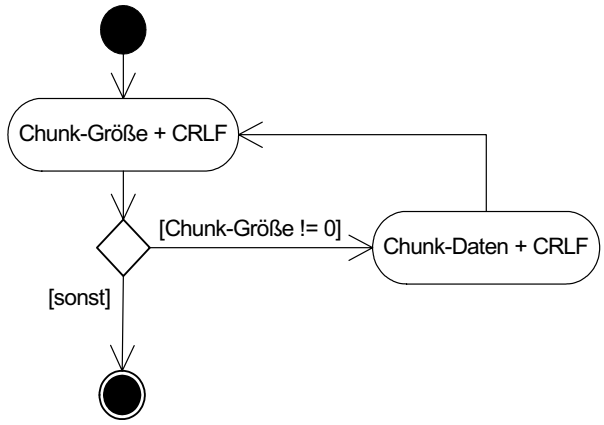


Abbildung 36: Die Übertragung in Chunks

Genau wie `normalFileTransfer` setzt auch `chunkedFileTransfer` das eigene Ganzzahlen-Format `CHugeNumber` aus Rezept 119 ein.

```
bool CHttpConnection::chunkedFileTransfer(const string &filename,
                                const CHttpResponse &response) {
  CBTRACELN("Uebertragung in Chunks beginnt...");

/*
** Datei auf lokalem Datenträger zum Schreiben öffnen
*/
  CFile file;
  if(!file.open(filename,true,true)) {
    m_lastError="FEHLER! Lokale Datei konnte nicht zum \
Schreiben geoeffnet werden!";
    CBTRACELN("FEHLER! Lokale Datei konnte nicht zum \
```

Listing 424: Die Methode chunkedFileTransfer

```
Schreiben geoeffnet werden!");
    return(false);
  }
  CMemory<char> buf(10000);
  unsigned int amount;
  CHugeNumber totallyRead, bufsize("10000");

/*
** Größe des ersten Chunks einlesen
*/
  CHugeNumber length=hexToValue(m_iconn->receiveLine());

/*
** Solange eingelesene Chunk-Größe != 0
*/
  while(length!=0) {
    CBTRACELN("Chungroesse : "+length.getAsPointedNumber()+
              " Bytes");
    do {

/*
** Darauf achten, dass nicht mehr Daten empfangen
** werden, als Chunk groß ist
*/
      if(length>=bufsize)
        amount=m_iconn->receive(buf,10000);
      else
        amount=m_iconn->receive(buf,length.getAsUnsignedLong());

      file.writeBlock(buf,amount);
      totallyRead+=static_cast<unsigned long>(amount);
      length-=static_cast<unsigned long>(amount);
      CBTRACE("Bisher uebertragen: "+
              totallyRead.getAsPointedNumber()+" Bytes\r");
      if((totallyRead==0)&&(length!=0)) {
        m_lastError="chunkedTransfer: Übertragung abgebrochen";
        CBTRACE("chunkedTransfer: Übertragung abgebrochen");
        return(false);
      }

    } while(length!=0);
```

Listing 424: Die Methode chunkedFileTransfer (Forts.)

```
    CBTRACELN("");

/*
** CRLF hinter Chunk-Daten einlesen
*/
    m_iconn->receiveLine();

/*
** Größe des nächsten Chunks einlesen
*/
    length=hexToValue(m_iconn->receiveLine());
  }

  CBTRACELN("\nErfolg!");
  return(true);
}
```

Listing 424: Die Methode chunkedFileTransfer (Forts.)

Die Klasse `CHttpConnection` finden Sie auf der CD in den `CHttpConnection`-Dateien.

Wir wollen zur Demonstration des Übertragungsvorgangs einmal die Startseite von Addison-Wesley laden:

```
#include "CHttpConnection.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  CSpecificINetConn::initialize();

  CUrl url("http://www.addison-wesley.de");
  CHttpConnection http;
  http.getFile(url,"start.html");
}
```

Listing 425: Ein Beispiel zu getFile

Die `main`-Funktion finden Sie auf der CD unter *Buchdaten\Beispiele\main0510.cpp*. `getFile` erzeugt aus der URL folgenden Request und sendet ihn an den Server:

```
GET / HTTP/1.1
Host: www.addison-wesley.de
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
```

Der Server von Addison-Wesley antwortet wie folgt:

```
HTTP/1.1 302 Object moved
Server: Microsoft-IIS/5.0
Date: Tue, 08 Jul 2003 21:49:31 GMT
Location: http://www.addison-wesley.de/main/main.asp?page=home/default
Content-Length: 181
Content-Type: text/html
Set-Cookie: ASPSESSIONIDGQQQGMSU=GMNNGCCADONPPBPFDIIJECJM; path=/
Cache-control: private
```

An der Kopf-Zeile, und dort speziell am Status-Code, erkennt man, dass der Server uns den tatsächlichen Ort der angeforderten Datei mitteilt.

`getFile` ruft sich selbst mit der neuen Location auf und sendet diesen Request:

GET /main/main.asp?page=home/default HTTP/1.1

Host: www.addison-wesley.de

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)

Nun antwortet der Server mit einem OK-Status:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 08 Jul 2003 22:07:59 GMT
pragma: no-cache
pragma: no-cache
Content-Length: 36536
Content-Type: text/html
Expires: Tue, 08 Jul 2003 22:06:58 GMT
```

Nun müssen nur noch die 36536 Bytes der gewünschten Ressource geladen und auf den lokalen Datenträger gespeichert werden.

74 Wie kann ein HEAD-Request an einen HTTP-Server gesendet werden?

Im Gegensatz zu einem GET-Request antwortet der Server auf einen HEAD-Request, ohne die angeforderte Ressource mitzusenden.

Wir erweitern dazu die Klasse `CHttpConnection` um eine Methode `getHeadHeader`, die eine URL zur Formulierung des Requests und einen Vektor zur Aufnahme der Server-Antwort übergeben bekommt.

```
bool CHttpConnection::getHeadHeader(const CUrl &url,
                                    vector<string> &header) {
  CBTRACELN("\ngetHeadHeader\n--------------------------------");

/*
** Head-Request formulieren
*/
  CHttpRequest request;
  request.setFromUrl(url);
  request.setRequestType(CHttpRequest::rtHEAD);
  request.setUserAgent(CHttpRequest::UAExplorer6);

/*
** Verbindung aufbauen
*/
  if(!connect(url)) {
    m_lastError="Fehler bei connect";
    CBTRACELN("Fehler bei connect");
    return(false);
  }

/*
** Request senden
*/
  if(!sendRequest(request)) {
    m_lastError="Fehler bei sendRequest";
    CBTRACELN("Fehler bei sendRequest");
```

Listing 426: Die Methode getHeadHeader von CHttpConnection

```
    return(false);
  }

/*
** Antwort empfangen
*/
  if(!receiveResponse(header)) {
    m_lastError="Fehler bei receiveResponse";
    CBTRACELN("Fehler bei receiveResponse");
    return(false);
  }

  CBTRACELN("Erfolg!");
  return(true);
}
```

Listing 426: Die Methode getHeadHeader von CHttpConnection (Forts.)

Zusätzlich wird die Methode noch überladen, um die Server-Antwort in Form eines `CHttpResponse`-Objektes zu erhalten:

```
bool CHttpConnection::getHeadHeader(const CUrl &url,
                                    CHttpResponse &header) {
  vector<string> vr;
  if(!getHeadHeader(url, vr))
    return(false);
  header.extract(vr);
  return(true);
}
```

Listing 427: Die überladene Methode getHeadHeader

Die Klasse `CHttpConnection` finden Sie auf der CD in den `CHttpConnection`-Dateien.

75 Wie kann ein TRACE-Request an einen HTTP-Server gesendet werden?

Ein TRACE-Request veranlasst alle beteiligten Server und Proxies, sich im `Via`-Header zu »verewigen«. Der `Via`-Header wird dann mit der Server-Antwort an den Client zurückgeschickt.

Um die Anzahl der einzelnen Stationen zu begrenzen, wird der Request im Normalfall mit dem `Max-Forwards`-Header ergänzt.

Die Methode `getTraceHeader` von `CHttpConnection` besitzt drei Parameter:

- `url` – die URL für den TRACE-Request
- `header` – ein Vektor, der die Server-Antwort aufnimmt
- `maxforwards` – der im Header `Max-Forwards` angegebene Wert

```
bool CHttpConnection::getTraceHeader(const CUrl &url,
                                     vector<string> &header,
                                     int maxforwards) {
  CBTRACELN("\ngetTraceHeader\n------------------------------");

/*
** Head-Request formulieren
*/
  CHttpRequest request;
  request.setFromUrl(url);
  request.setRequestType(CHttpRequest::rtTRACE);
  request.setUserAgent(CHttpRequest::UAExplorer6);
  request.setMaxForwards(maxforwards);

/*
** Verbindung aufbauen
*/
  if(!connect(url)) {
    m_lastError="Fehler bei connect";
    CBTRACELN("Fehler bei connect");
    return(false);
  }

/*
** Request senden
*/
  if(!sendRequest(request)) {
    m_lastError="Fehler bei sendRequest";
    CBTRACELN("Fehler bei sendRequest");
    return(false);
  }
```

Listing 428: Die Methode getTraceHeader von CHttpConnection

```
/*
** Antwort empfangen
*/
  if(!receiveResponse(header)) {
    m_lastError="Fehler bei receiveResponse";
    CBTRACELN("Fehler bei receiveResponse");
    return(false);
  }

  CBTRACELN("Erfolg!");
  return(true);
}
```

Listing 428: Die Methode getTraceHeader von CHttpConnection (Forts.)

Zur einfacheren Handhabung wird die Methode mit einem `CHttpResponse`-Objekte als Parameter überladen.

```
bool CHttpConnection::getTraceHeader(const CUrl &url,
                                     CHttpResponse &header,
                                     int maxforwards) {
  vector<string> vr;
  if(!getTraceHeader(url, vr, maxforwards))
    return(false);
  header.extract(vr);
  return(true);
}
```

Listing 429: Die überladene Methode getTraceHeader

Die Klasse `CHttpConnection` finden Sie auf der CD in den `CHttpConnection`-Dateien.

76 Wie loggt man sich auf einem FTP-Server ein?

Für das Einloggen auf einem FTP-Server entwerfen wir eine eigene Klasse `CFtpConnection`.

CFtpConnection

`CFtpConnection` wickelt sämtliche Internet-Kommunikation über die Klasse `CInternetConnection` ab, die auf der Internet-Funktionalität von `CSpecificINetCon` aus Rezept 69 basiert.

Klassendefinition

Im Gegensatz zum HTTP-Protokoll, wo die Kommunikation zwischen Clients im Allgemeinen auf einen Request mit anschließender Antwort beschränkt ist, besteht das FTP-Protokoll aus einer aufwändigeren Kommunikation. Der Client loggt sich auf dem Server ein und gelangt häufig erst durch das Absetzen mehrerer Befehle an die gewünschte Ressource.

Aus diesem Grund steht ein Objekt der Klasse `CFtpConnection` für die bestehende Verbindung zu einem FTP-Server. Die Verbindung wird durch die Erzeugung des Objekts hergestellt und besteht, bis das Objekt gelöscht wird[2].

Die Klasse besitzt folgende Attribute:

- `AnonymousName` – ein statisches Attribut, welches den Namen für einen anonymen Login beinhaltet
- `AnonymousPwd` – ein statisches Attribut mit dem Passwort für einen anonymen Login
- `m_lastError` – enthält den zuletzt aufgetretenen Fehler, im Allgemeinen die letzte vom Server gesendete Nachricht.
- `m_rootDir` – das Hauptverzeichnis des FTP-Servers
- `m_currentDir` – das Verzeichnis auf dem Server, in dem sich der Client gerade befindet.
- `m_iconn` – ein Zeiger auf das `CInternetConnection`-Objekt, über das die TCP-Kommunikation abgewickelt wird.

```
class CFtpConnection {
private:
  typedef std::pair<std::string, std::string> ServerReply;
  typedef std::pair<std::string, unsigned int> Address;

  static const std::string AnonymousName;
  static const std::string AnonymousPwd;

  std::string m_lastError;
  std::string m_rootDir;
  std::string m_currentDir;
```

Listing 430: Die Klassendefinition von CFtpConnection

2. Dies alles natürlich nur unter der Voraussetzung, dass keine Fehler auftreten.

```
  CInternetConnection *m_iconn;

public:
  enum FtpType {A='A', I='I', B='B'};
};
```

Listing 430: Die Klassendefinition von CFtpConnection (Forts.)

Im Folgenden werden die statischen Attribute initialisiert:

```
const string CFtpConnection::AnonymousName="anonymous";
const string CFtpConnection::AnonymousPwd="test";
```

Listing 431: Initialisierung der statischen Attribute

Konstruktoren

Der erste Konstruktor besitzt folgende Attribute:

- `host` – enthält die Host-Adresse des Ziel-Servers.
- `port` – beinhaltet den Port, über den Kontakt mit dem Server aufgenommen werden soll (Standardwert: 21).
- `newlogin` – boolescher Wert, der darüber entscheidet, ob es sich um einen neuen Login handelt (warten auf Welcome-Nachricht) oder um einen Login zwecks Datenübertragung.
- `retries` – gibt an, wie oft bei Misserfolg die Verbindungsaufnahme wiederholt werden soll (Standard: 5).

```
CFtpConnection::CFtpConnection(const string &host,
                               unsigned int port,
                               bool newlogin,
                               int retries)
: m_iconn(0) {

  int trynum=1;
  bool success=false;
  do {
    m_iconn=0;
```

Listing 432: Ein Konstruktor von CFtpConnection

```
    CBTRACELN("\nCFtpConnection Konstruktor...\n---------------");
    CBTRACELN("Verbindungsaufbau (Versuch "+toString(trynum)+")");

/*
** Verbindung mit Server herstellen
*/
    m_iconn=new CInternetConnection(host, port);

/*
** Steht Verbindung?
*/
    if(m_iconn->isAvailable()) {

/*
** Bei neuem Login auf Welcome-Nachricht warten
*/
      if(newlogin) {
        try {
          ServerReply reply=waitForCmdGroup(2);
          CBTRACELN("Empfangen: "+rebuildReply(reply));
          if(reply.first!="220") {
            CBTRACELN("Abbruch! Kein 220-Code bei Anmeldung");
          }
          else
            success=true;
        }
        catch(...) {
          CBTRACELN("Abbruch bei Empfang der Welcome-Meldung!");

        }
      }
      else
        success=true;
    }

/*
** Verbindung fehlgeschlagen?
** => CInternetConnection-Objekt löschen
*/
    if(!success)
      delete(m_iconn);
```

Listing 432: Ein Konstruktor von CFtpConnection (Forts.)

```
  } while((!success)&&(retries>trynum++));

  if(success) {
    CBTRACELN("Erfolg!");
  }

/*
** Alle Versuche durch und immer noch kein Erfolg
*/
  else {
    CBTRACELN("Endgültiger Abbruch!");
    throw internet_error("CFtpConnection::CFtpConnection");
  }
}
```

Listing 432: Ein Konstruktor von CFtpConnection (Forts.)

Der zweite Konstruktor ist ähnlich aufgebaut, nur dass er sich Host und Port aus einem `CUrl`-Objekt holt:

```
CFtpConnection::CFtpConnection(const CUrl &url,
                               bool newlogin,
                               int retries)
: m_iconn(0) {

/*
** Host und Port aus URL holen
*/
  string host=url.getDomain();
  unsigned int port;
  if(url.getPort()!="")
    port=atol(url.getPort().c_str());
  else
    port=21;

  int trynum=1;
  bool success=false;
  do {
    m_iconn=0;
    CBTRACELN("\nCFtpConnection Konstruktor...\n----------------");
```

Listing 433: Ein Konstruktor mit einem CUrl-Objekt

```
    CBTRACELN("Verbindungsaufbau (Versuch "+toString(trynum)+")");

/*
** Verbindung mit Server herstellen
*/
    m_iconn=new CInternetConnection(host, port);

/*
** Steht Verbindung?
*/
    if(m_iconn->isAvailable()) {

/*
** Bei neuem Login auf Welcome-Nachricht warten
*/
      if(newlogin) {
        try {
          ServerReply reply=waitForCmdGroup(2);
          CBTRACELN("Empfangen: "+rebuildReply(reply));
          if(reply.first!="220") {
            CBTRACELN("Abbruch! Kein 220-Code bei Anmeldung");
          }
          else
            success=true;
        }
        catch(...) {
          CBTRACELN("Abbruch bei Empfang der Welcome-Meldung!");

        }
      }
      else
        success=true;
    }

/*
** Verbindung fehlgeschlagen?
** => CInternetConnection-Objekt löschen
*/
    if(!success)
      delete(m_iconn);
  } while((!success)&&(retries>trynum++));
```

Listing 433: Ein Konstruktor mit einem CUrl-Objekt (Forts.)

```
  if(success) {
    CBTRACELN("Erfolg!");
  }

/*
** Alle Versuche durch und immer noch kein Erfolg
*/
  else {
    CBTRACELN("Endgültiger Abbruch!");
    throw internet_error("CFtpConnection::CFtpConnection");
  }
}
```

Listing 433: Ein Konstruktor mit einem CUrl-Objekt (Forts.)

Grundsätzlich lässt sich nicht ermitteln, aus wie vielen Zeilen die Willkommens-Nachricht besteht. Um ein Blockieren der `read`-Methode zu verhindern, dürfen wir keine Daten vom Server empfangen, wenn er keine sendet.

Aus diesem Grund lesen wir nur eine Zeile der Willkomens-Nachricht ein und überlesen dann im weiteren Login-Verlauf eventuell vorhandene zusätzliche Zeilen.

Destruktor

Der Destruktor gibt lediglich das eventuell vorhandene `CInternetConnection`-Objekt frei:

```
CFtpConnection::~CFtpConnection() {
  if(m_iconn)
    delete(m_iconn);
}
```

Listing 434: Der Destruktor von CFtpConnection

splitReply

Diese Methode teilt die Antwort des Servers in den Antwort-Code und den Antwort-Text auf und liefert das Ergebnis als Paar zurück:

```
CFtpConnection::ServerReply
CFtpConnection::splitReply(const string &line) const {
```

Listing 435: die Hilfsmethode splitReply

```
  ServerReply reply;
  if(line.size()>4) {
    reply.first=line.substr(0,3);
    reply.second=line.substr(4);
  }
  return(reply);
}
```

Listing 435: die Hilfsmethode splitReply (Forts.)

rebuildReply

Mit `rebuildReply` kann ein `ServerReply`-Objekt – zum Beispiel zu Zwecken der Ausgabe – wieder zu einem String zusammengesetzt werden.

```
string CFtpConnection::rebuildReply(
       const CFtpConnection::ServerReply &reply) const {
  return(reply.first+" "+reply.second);
}
```

Listing 436: Die Hilfsmethode rebuildReply

waitForAnyCmd

Die Methode `waitForAnyCmd` wartet auf eine beliebige Antwort vom Server und liefert sie als `ServerReply`-Objekt zurück.

```
CFtpConnection::ServerReply CFtpConnection::waitForAnyCmd() {
  ServerReply reply=splitReply(m_iconn->receiveLine());

/*
** Keine Server-Antwort?
** => Ausnahme werfen
*/
  if(reply.first=="") {
    CBTRACELN("waitForAnyCmd: Keine Server-Antwort! => \
Ausnahme werfen");
    throw internet_error("waitForAnyCmd: Keine Server-Antwort");
  }
```

Listing 437: Die Hilfsmethode waitForAnyCmd

```
  return(reply);
}
```

Listing 437: Die Hilfsmethode waitForAnyCmd (Forts.)

waitForSpecificCmd

Diese Hilfsmethode empfängt so lange Zeilen vom Server, bis der Server den gewünschten Antwort-Code sendet.

```
CFtpConnection::ServerReply
CFtpConnection::waitForSpecificCmd(int cmd) {
  ServerReply reply;
  do {
    reply=splitReply(m_iconn->receiveLine());

/*
** Keine Server-Antwort?
** => Ausnahme werfen
*/
    if(reply.first=="") {
      CBTRACELN("waitForSpecificCmd: Keine Server-Antwort! => \
Ausnahme werfen");
      throw internet_error("waitForSpecificCmd: Keine Server\
-Antwort");
    }

#ifdef CBTR
    if(atoi(reply.first.substr(0,3).c_str())!=cmd)
      CBTRACELN("UEBERLESEN: "+rebuildReply(reply));
#endif

/*
** Einlesen wiederholen, bis gültige Antwort
** empfangen wurde
*/
  } while(atoi(reply.first.substr(0,3).c_str())!=cmd);
  return(reply);
}
```

Listing 438: Die Hilfsmethode waitForSpecificCmd

waitForCmdGroup

Die Methode `waitForCmdGroup` wartet auf eine Server-Antwort mit dem Antwort-Code einer bestimmten Gruppe. Zur Gruppen-Identifikation wird die erste Ziffer des Antwort-Codes verwendet:

```
CFtpConnection::ServerReply
CFtpConnection::waitForCmdGroup(int group) {
  ServerReply reply;
  do {
    reply=splitReply(m_iconn->receiveLine());

/*
** Keine Server-Antwort?
** => Ausnahme werfen
*/
    if(reply.first=="") {
      CBTRACELN("waitForCmdGroup: Keine Server-Antwort! => Ausnahme
\werfen");
      throw internet_error("waitForCmdGroup: Keine Server-Antwort");
    }

#ifdef CBTR
    if(atoi(reply.first.substr(1,1).c_str())!=group)
      CBTRACELN("UEBERLESEN: "+rebuildReply(reply));
#endif

/*
** Einlesen wiederholen, bis gültige Antwort
** empfangen wurde
*/
  } while(atoi(reply.first.substr(1,1).c_str())!=group);
  return(reply);
}
```

Listing 439: Die Hilfsmethode waitForCmdGroup

login

Der Login basiert im Wesentlichen auf drei Befehlen: USER um den Usernamen mitzuteilen, PASS, um ein eventuelles Passwort zu übermitteln, und in seltenen Fällen ACCT, um einen Account zu spezifizieren.

Während dieses Login-Vorgangs teilt uns die erste Ziffer des dreistelligen Antwort-Codes mit, ob der Prozess erfolgreich war oder nicht. Abbildung 37 stellt den Ablauf grafisch dar.

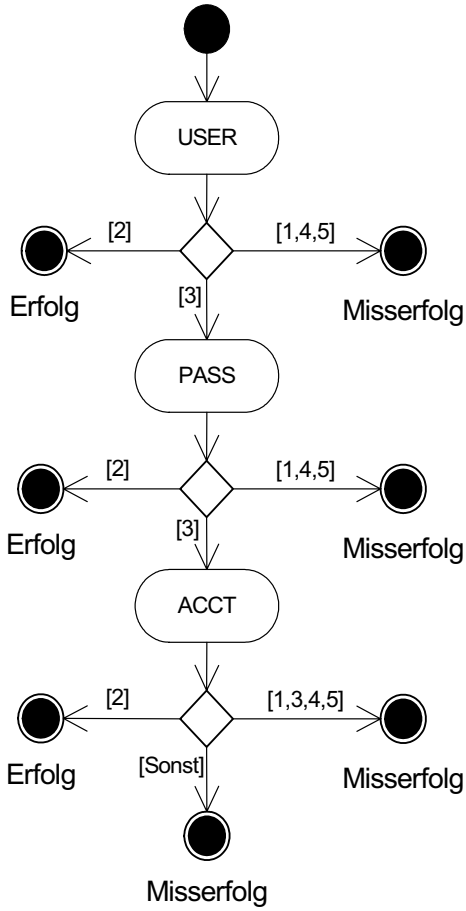


Abbildung 37: Der login-Vorgang über das FTP-Protokoll

Dieser Ablauf spiegelt sich in der folgenden Methode wider:

```
bool CFtpConnection::login(const string &username,
                           const string &pwd,
                           const string &account) {
  CBTRACELN("\nlogin...\n-------------------------------------");
```

Listing 440: Die Methode login von CFtpConnection

```
/*
** Usernamen senden
*/
  m_iconn->sendLineCRLF("USER "+username);
  CBTRACELN("Gesendet : USER "+username);
  ServerReply reply=waitForCmdGroup(3);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** User ist nicht bekannt o.Ä.
** => Abbruch
*/
  if((reply.first[0]=='1')||
     (reply.first[0]=='4')||
     (reply.first[0]=='5')) {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }

/*
** Username allein reicht für Login
** => Erfolg!
*/
  if(reply.first[0]=='2') {
    CBTRACELN("Erfolg!");
    m_currentDir=m_rootDir=getCurrentDir();
    return(true);
  }

/*
** Es wird das Passwort benötigt
*/
  m_iconn->sendLineCRLF("PASS "+pwd);
  CBTRACELN("Gesendet : PASS "+pwd);
  reply=waitForCmdGroup(3);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** User oder Pwd inkorrekt
```

Listing 440: Die Methode login von CFtpConnection (Forts.)

```
** => Abbruch
*/
  if((reply.first[0]=='1')||
     (reply.first[0]=='4')||
     (reply.first[0]=='5')) {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }

/*
** Passwort gültig und ausreichend
** => Erfolg!
*/
  if(reply.first[0]=='2') {
    CBTRACELN("Erfolg!");
    m_currentDir=m_rootDir=getCurrentDir();
    return(true);
  }

/*
** Es wird ein Account benötigt
*/
  m_iconn->sendLineCRLF("ACCT "+account);
  CBTRACELN("Gesendet : ACCT "+account);
  reply=waitForCmdGroup(3);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Account o.k.
** => Erfolg!
*/
  if(reply.first[0]=='2') {
    CBTRACELN("Erfolg!");
    m_currentDir=m_rootDir=getCurrentDir();
    return(true);
  }

/*
** Login fehlgeschlagen
*/
```

Listing 440: Die Methode login von CFtpConnection (Forts.)

```
  m_lastError=rebuildReply(reply);
  CBTRACELN("Abbruch!");
  return(false);
}
```

Listing 440: Die Methode login von CFtpConnection (Forts.)

War der Login-Vorgang erfolgreich, wird vom Server das aktuelle Verzeichnis ermittelt, was in diesem Fall dem Hauptverzeichnis entspricht.

Damit beim Login auch ein `CUrl`-Objekt eingesetzt werden kann, wird die `login`-Methode überladen:

```
bool CFtpConnection::login(const CUrl &url, const string &account) {

/*
** Username und Passwort aus URL holen
*/
  string username=url.getUsername();
  if(username=="")
    username=AnonymousName;

  string password=url.getPassword();
  if(password=="")
    password=AnonymousPwd;

/*
** Echte login-Methode aufrufen
*/
  return(login(username,password,account));

}
```

Listing 441: Die Methode login für CUrl-Objekte

getCurrentDir

Die Methode `getCurrentDir` ermittelt über den FTP-Befehl PWD das Server-Verzeichnis, in dem sich der Client augenblicklich befindet.

```
string CFtpConnection::getCurrentDir() {
  CBTRACELN("\ngetCurrentDir...\n---------------------------");

/*
** Befehl senden
*/
  m_iconn->sendLineCRLF("PWD");
  CBTRACELN("Gesendet : PWD");

/*
** Auf spezielle Antwort (Code 257) auf
** PWD warten
*/
  ServerReply reply=waitForSpecificCmd(257);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Das Verzeichnis steht im Antwort-Text zwischen
** zwei doppelten Anführungszeichen
*/
  string::size_type beg=reply.second.find("\"")+1;
  string::size_type len=reply.second.find("\"",beg)-beg;
  string dir=reply.second.substr(beg,len);
  CBTRACELN("Extrahiertes Verzeichnis: "+dir);

  return(dir);
}
```

Listing 442: Die Methode getCurrentDir von CFtpConnection

Die Klasse `CFtpConnection` finden Sie auf der CD in den `CFtpConnection`-Dateien.

77 Wie kann eine Datei von einem FTP-Server übertragen werden?

Um eine Datei von einem FTP-Server herunterzuladen, muss sich der Client zunächst auf dem Server einloggen. Diese Aufgabe meistert bereits die Klasse `CFtp-Connection` aus dem vorigen Rezept.

changeDir

Der nächste Schritt besteht darin, auf dem Server in das Verzeichnis der gewünschten Datei zu wechseln.

Diese Aufgabe erledigen wir mit der Methode `changeDir`, die den FTP-Befehl `CWD` zum Wechseln des Verzeichnisses einsetzt.

```
bool CFtpConnection::changeDir(const string &path) {
  CBTRACELN("\nchangeDir...\n--------------------------------");

/*
** CWD-Befehl senden
*/
  m_iconn->sendLineCRLF("CWD "+path);
  CBTRACELN("Gesendet : CWD "+path);
  ServerReply reply=waitForCmdGroup(5);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Informiert Status-Code nicht über Erfolg?
** => Abbruch
*/
  if(reply.first[0]!='2') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }

/*
** Zur Bestätigung aktuelles Verzeichnis holen
*/
  CBTRACELN("Erfolg!");
  m_currentDir=getCurrentDir();
  return(true);
}
```

Listing 443: Die Methode changeDir von CFtpConnection

retrieveFile

`retrieveFile` überträgt nun die gewünschte Datei vom Server.

Ihr wird der Name der Datei auf dem Server und die Zieldatei auf dem lokalen Datenträger übergeben.

Zunächst setzt die Methode die Übertragungsart auf binär. Anschließend wird in den passiven Modus gewechselt.

Grundsätzlich bietet das FTP-Protokoll zwei Möglichkeiten, Daten vom Server zum Client zu übertragen. Entweder nimmt der Server eigenständig mit der empfangenden Einheit des Clients Kontakt auf und sendet die Daten oder – und das ist der hier eingesetzte passive Modus – der Server stellt eine Möglichkeit zur Verbindungsaufnahme bereit und der Client stellt die Verbindung her.

Nachdem sich der Server im passiven Modus befindet, nimmt unsere Methode mit einem neuen `CFtpConnection`-Objekt Kontakt für die Übertragung der Daten auf und empfängt so lange, bis der Server die Datenverbindung trennt.

```
bool CFtpConnection::retrieveFile(const string &serverFile,
                                  const string &localFile) {

/*
** Übertragungsart auf Binär setzen
*/
  setType(I);

/*
** In den passiven Modus wechseln
*/
  Address paddr=setPassive();
  CBTRACELN("\nretrieveFile...\n------------------------------");

/*
** Lokale Datei zum Schreiben öffnen
*/
  CFile file;
  if(!file.open(localFile,true,true)) {
    CBTRACELN("FEHLER! Lokale Datei konnte nicht zum \
Schreiben geoeffnet werden!");
    return(false);
  }

/*
** Retrieve-Befehl senden
*/
  m_iconn->sendLineCRLF("RETR "+serverFile);
```

Listing 444: Die Methode retrieveFile von CFtpConnection

```
  CBTRACELN("Gesendet : RETR "+serverFile);
  CFtpConnection receiver(paddr.first, paddr.second,false);
  ServerReply reply=waitForAnyCmd();
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Werden keine Daten gesendet?
** => Abbruch!
*/
  if(reply.first[0]!='1') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }
  CMemory<char> buf(10000);
  unsigned int amount;
  CHugeNumber totallyRead;

/*
** Daten empfangen, bis Server die
** Datenverbindung abbricht
*/
  do {
    amount=receiver.m_iconn->receive(buf,10000);
    file.writeBlock(buf,amount);
    totallyRead+=static_cast<unsigned long>(amount);
    CBTRACE("Bisher uebertragen: "+
            totallyRead.getAsPointedNumber()+" Bytes\r");
  } while(amount);

  CBTRACELN("");
  reply=waitForCmdGroup(2);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Übertragung nicht korrekt beendet?
** => Abbruch!
*/
  if(reply.first[0]!='2') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
```

Listing 444: Die Methode retrieveFile von CFtpConnection (Forts.)

```
    return(false);
  }

  CBTRACELN("Erfolg!");
  getCurrentDir();
  return(true);
}
```

Listing 444: Die Methode retrieveFile von CFtpConnection (Forts.)

Damit eine Datei auch unabhängig von einem bestehenden `CFtpConnection`-Objekt übertragen werden kann, implementieren wir eine statische Variante von `retrieveFile`, die ein `CUrl`-Objekt zur Definition der gewünschten Ressource übergeben bekommt.

Im Gegensatz zur vorigen `retrieveFile`-Methode bekommt die neue Version zur Spezifizierung des Ziels auf dem lokalen Datenträger nur einen Pfad übergeben, der keinen abschließenden / enthalten darf. Für die Zieldatei wird als Dateiname der aus dem `CUrl`-Objekt extrahierte Original-Name verwendet.

```
bool CFtpConnection::retrieveFile(const CUrl &url,
                                  const string &localPath) {
  CBTRACELN("\nretrieveFile...\n----------------------------");

/*
** Pfade und Dateinamen extrahieren
*/
  string serverPath=url.getPath();
  if(serverPath=="") {
    CBTRACELN("Abbruch! (kein gueltiger Pfad)");
    return(false);
  }
  string localFile=localPath;
  string fileName;
  string::size_type spos=serverPath.rfind("/");
  if(spos!=string::npos) {
    fileName=serverPath.substr(spos+1);
    localFile+="/"+fileName;
    serverPath="/"+serverPath.substr(0,spos+1);
  }
```

Listing 445: Die statische Methode retrieveFile

```
  else {
    fileName=serverPath;
    localFile+="/"+fileName;
    serverPath="/";
  }

  CBTRACELN("Dateiname   : "+fileName);
  CBTRACELN("FTP-Pfad    : "+serverPath);
  CBTRACELN("Lokaler Pfad: "+localFile);

/*
** Port aus CUrl-Objekt extrahieren. Wenn keiner
** vorhanden, dann Port 21 nehmen
*/
  unsigned int port;
  if(url.getPort()!="")
    port=atol(url.getPort().c_str());
  else
    port=21;

/*
** Usernamen aus CUrl-Objekt extrahieren. Ist keiner vorhanden,
** wird der in AnonymousName definierte verwendet.
*/
  string username=url.getUsername();
  if(username=="")
    username=AnonymousName;

/*
** Passwort aus CUrl-Objekt extrahieren. Ist keins vorhanden,
** wird das in AnonymousPwd definierte verwendet.
*/
  string password=url.getPassword();
  if(password=="")
    password=AnonymousPwd;

  try {

/*
** Verbindungsaufbau zu Server
*/
```

Listing 445: Die statische Methode retrieveFile (Forts.)

```
    CFtpConnection ftp(url.getDomain(), port);

/*
** Einloggen
*/
    if(!ftp.login(username, password)) {
      CBTRACELN("RetrieveFile: Login fehlgeschlagen");
      return(false);
    }

/*
** In das Verzeichnis der gewünschten Ressource wechseln
*/
    if(!ftp.changeDir(serverPath)) {
      CBTRACELN("RetrieveFile: Verzeichniswechsel fehlgeschlagen");
      return(false);
    }

/*
** Datei empfangen
*/
    if(!ftp.retrieveFile(fileName,localFile)) {
      CBTRACELN("RetrieveFile: Dateiuebertragung fehlgeschlagen");
      return(false);
    }
  }
  catch(internet_error) {
    CBTRACELN("RetrieveFile fehlgeschlagen (Exception!)");
    return(false);
  }

  CBTRACELN("retrieveFile Erfolg!");
  return(true);
}
```

Listing 445: Die statische Methode retrieveFile (Forts.)

Im Folgenden werden die von `retrieveFile` verwendeten Methoden erläutert.

setType

Mit der Methode `setType` wird die Übertragungsart des Servers bestimmt. Für unsere Zwecke sind nur Text- und Binärmodus interessant.

```
bool CFtpConnection::setType(CFtpConnection::FtpType type) {
  CBTRACELN("\nsetType...\n--------------------------------");

/*
** Type senden
*/
  m_iconn->sendLineCRLF(static_cast<string>("TYPE ")+
                        static_cast<char>(type));
  CBTRACELN(static_cast<string>("Gesendet : TYPE ")+
            static_cast<char>(type));
  ServerReply reply=waitForCmdGroup(0);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Type erfolgreich gesetzt?
** => Erfolg!
*/
  if(reply.first[0]=='2') {
    CBTRACELN("Erfolg!");
    return(true);
  }

/*
** setType fehlgeschlagen
*/
  m_lastError=rebuildReply(reply);
  CBTRACELN("Abbruch!");
  return(false);
}
```

Listing 446: Die Methode setType von CFtpConnection

setPassive

Wenn der Server erfolgreich in den passiven Modus versetzt wurde, liefert er eine Rückmeldung, die wie folgt aussehen kann:

```
227 Entering Passive Mode (192,168,0,1,72,96)
```

Die Zahlen in den runden Klammern definieren Adresse und Port für die Datenübertragung. Die ersten vier Ziffern bilden dabei die Komponenten einer IP-Adresse. Es müssen lediglich Punkte dazwischengesetzt werden.

Die letzten beiden Zahlen definieren den Port in Form eines High- und Low-Bytes.

```
CFtpConnection::Address CFtpConnection::setPassive() {
  CBTRACELN("\nsetPassive...\n--------------------------------");

/*
** Passive-Befehl senden
*/
  m_iconn->sendLineCRLF("PASV");
  CBTRACELN("Gesendet : PASV");
  ServerReply reply=waitForCmdGroup(2);
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Passive-Modus nicht eingestellt?
** => Abbruch!
*/
  if(reply.first[0]!='2') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    throw internet_error("CFtpConnection::setPassive: "+
                         rebuildReply(reply));
  }

/*
** Adresse und Port extrahieren
*/
  Address addr;
  string info=reply.second.substr(reply.second.find("(")+1);
  info=info.substr(0,info.find(")"))+",";
  CBTRACELN("Extrahierte Information: "+info);

  int vals[6];
  string::size_type beg=0,end=info.find(",");
  for(int i=0;i<6;i++) {
    vals[i]=atoi(info.substr(beg,end-beg+1).c_str());
    beg=end+1;
```

Listing 447: Die Methode setPassive von CFtpConnection

```
    end=info.find(",",beg);
  }

/*
** Adresse und Port in "nutzbares" Format übertragen
*/
  addr.first=toString(vals[0])+"."+
             toString(vals[1])+"."+
             toString(vals[2])+"."+
             toString(vals[3]);
  addr.second=vals[4]*256+vals[5];
  CBTRACELN("Passive Adresse: "+addr.first);
  CBTRACELN("Passiver Port  : "+toString(addr.second));

  CBTRACELN("Erfolg!");
  return(addr);
}
```

Listing 447: Die Methode setPassive von CFtpConnection (Forts.)

Die Klasse `CFtpConnection` finden Sie auf der CD in den `CFtpConnection`-Dateien.

78 Wie kann ein Verzeichnis auf einem FTP-Server ausgelesen werden?

Zur Lösung dieser Aufgabe greifen wir auf die in den vorgehenden Rezepten entwickelte Klasse `CFtpConnection` zurück und erweitern sie um die statische Methode `getDir`.

getDir

Sie entimmt die benötigten Informationen aus einem `CUrl`-Objekt und speichert die Einträge als `CFileInfo`-Objekte in das übergebene `filelist_type`-Objekt.

Das tatsächliche Einlesen der Einträge vom Server übernimmt die weiter unten aufgeführte Methode `list`.

```
bool CFtpConnection::getDir(const CUrl &url,
                            CFileInfo::filelist_type &files) {
  CBTRACELN("\ngetDir...\n----------------------------------");
```

Listing 448: Die statische Methode getDir von CFtpConnection

```
  string serverPath="/"+url.getPath();

  CBTRACELN("FTP-Pfad    : "+serverPath);

/*
** Port aus CUrl-Objekt extrahieren. Wenn keiner
** vorhanden, dann Port 21 nehmen
*/
  unsigned int port;
  if(url.getPort()!="")
    port=atol(url.getPort().c_str());
  else
    port=21;

/*
** Usernamen aus CUrl-Objekt extrahieren. Ist keiner vorhanden,
** wird der in AnonymousName definierte verwendet.
*/
  string username=url.getUsername();
  if(username=="")
    username=AnonymousName;

/*
** Passwort aus CUrl-Objekt extrahieren. Ist keins vorhanden,
** wird das in AnonymousPwd definierte verwendet.
*/
  string password=url.getPassword();
  if(password=="")
    password=AnonymousPwd;

  try {

/*
** Verbindungsaufbau zu Server
*/
    CFtpConnection ftp(url.getDomain(),port);

/*
** Einloggen
*/
    if(!ftp.login(username, password)) {
```

Listing 448: Die statische Methode getDir von CFtpConnection (Forts.)

```
      CBTRACELN("getDir: Login fehlgeschlagen");
      return(false);
    }

/*
** In das auszulesende Verzeichnis wechseln
*/
    if(!ftp.changeDir(serverPath)) {
      CBTRACELN("getDir: Verzeichniswechsel fehlgeschlagen");
      return(false);
    }

/*
** Verzeichnis auslesen
*/
    if(!ftp.list(files)) {
      CBTRACELN("getDir: list fehlgeschlagen");
      return(false);
    }
  }
  catch(internet_error) {
    CBTRACELN("getDir fehlgeschlagen (Exception!)");
    return(false);
  }

  CBTRACELN("getDir Erfolg!");
  return(true);
}
```

Listing 448: Die statische Methode getDir von CFtpConnection (Forts.)

list

Die Methode `list` setzt den tatsächlichen FTP-Befehl `LIST` um und empfängt ähnlich wie `retrieveFile` die Verzeichnis-Einträge über eine eigene Datenverbindung. Dazu versetzt sie den Server zuvor in den passiven Modus.

Die einzelnen Einträge werden mit der später aufgeführten Methode `convertListEntryToFileInfo` in `CFileInfo`-Objekte umgewandelt.

```
bool CFtpConnection::list(CFileInfo::filelist_type &filelist) {

/*
** Type auf Text setzen
*/
  setType(A);

/*
** In den passiven Modus wechseln
*/
  Address paddr=setPassive();
  CBTRACELN("\nlist...\n-----------------------------------");

/*
** List-Befehl senden
*/
  m_iconn->sendLineCRLF("LIST");
  CBTRACELN("Gesendet : LIST");
  CFtpConnection receiver(paddr.first, paddr.second,false,10);
  ServerReply reply=waitForAnyCmd();
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Wird keine Datei-Liste gesendet?
** => Abbruch!
*/
  if(reply.first[0]!='1') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }

/*
** Dateiliste einlesen
*/
  string line;
  while((line=receiver.m_iconn->receiveLine())!="") {
    filelist.push_back(convertListEntryToFileInfo(line));
  }

  reply=waitForAnyCmd();
```

Listing 449: Die Methode list von CFtpConnection

```
  CBTRACELN("Empfangen: "+rebuildReply(reply));

/*
** Übertragung nicht korrekt beendet?
** => Abbruch!
*/
  if(reply.first[0]!='2') {
    m_lastError=rebuildReply(reply);
    CBTRACELN("Abbruch!");
    return(false);
  }

  CBTRACELN("Erfolg!");
  getCurrentDir();
  return(true);
}
```

Listing 449: Die Methode list von CFtpConnection (Forts.)

convertListEntryToFileInfo

Nachdem der Client einen `LIST`-Befehl abgesetzt hat, sendet der Server die einzelnen Verzeichnis-Einträge über eine eigenständige Verbindung an den Client.

Obwohl nicht garantiert, besitzt das Format heutzutage üblicherweise die gleiche Form wie der Unix-Befehl `ls -l`. Im Folgenden sehen Sie einen Beispiel-Eintrag:

```
drwxr-xr-x   1 ftp      ftp             0 Mar 15 21:26 Buch
```

Der erste Block definiert den Dateityp und die Zugriffsrechte. Steht an der ersten Position ein `d`, dann handelt es sich um ein Verzeichnis, normale Dateien haben dort ein – stehen.

Von den einzelnen Feldern wird für uns erst wieder die Dateigröße interessant, die im oberen Beispiel eines Verzeichnis-Eintrags 0 ist.

Hinter der Dateigröße steht das Datum, an dem die Datei das letzte Mal modifiziert wurde. Ist die Änderung länger als sechs Monate her, steht hinter dem Monat das Jahr der Änderung. Bei einem jüngeren Änderungsdatum ist anstelle des Jahres die Uhrzeit der Änderung aufgeführt.

Hinter dem Änderungsdatum steht dann der Datei- bzw. Verzeichnisname.

Die Methode `convertListEntryToFileInfo` extrahiert lediglich Dateityp, -größe und -name.

```
CFileInfo
CFtpConnection::convertListEntryToFileInfo(const string &line) {
  CBTRACELN(line);
  string::size_type p1=0,p2;

/*
** Zugriffsrechte und Dateityp extrahieren
*/
  while(line[p1]!=' ') p1++;
  string permissions=line.substr(0,p1);

/*
** Bis zur Dateigröße vorlaufen
*/
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
  while(line[p1]==' ') p1++;

/*
** Dateigröße extrahieren
*/
  p2=p1;
  while(line[p1]!=' ') p1++;
  CHugeNumber filesize(line.substr(p2,p1-p2));

/*
** Bis zum Dateinamen vorlaufen
*/
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
  while(line[p1]==' ') p1++;
  while(line[p1]!=' ') p1++;
```

Listing 450: Die Methode convertListEntryToFileInfo

```
  while(line[p1]==' ') p1++;

/*
** Dateinamen extrahieren
*/
  string name=line.substr(p1);

  CFileInfo fi;

/*
** Dateityp extrahieren
*/
  if(tolower(permissions[0])=='d')
    fi.m_type=CFileInfo::DIR;
  else if(tolower(permissions[0])=='-')
    fi.m_type=CFileInfo::FILE;

/*
** Informationen in einem CFileInfo-Objekt speichern
*/
  fi.m_name=name;
  fi.m_path=m_currentDir;
  fi.m_size=filesize.getAsFileSize();

#ifdef CBTR
  if(fi.m_type==CFileInfo::DIR)
    cout << "[DIR] ";
  cout << fi.m_name << " " << filesize.getAsPointedNumber() << endl;
#endif

  return(fi);
}
```

Listing 450: Die Methode convertListEntryToFileInfo (Forts.)

Die Klasse `CFtpConnection` finden Sie auf der CD in den `CFtpConnection`-Dateien.

79 Wie kann die komplette Verzeichnis-Struktur eines FTP-Servers ausgelesen werden?

Der Ansatz ist ähnlich dem aus dem vorhergehenden Rezept, wird jedoch um rekursive Aufrufe ergänzt.

getCompleteDir

Zunächst wird eine statische Methode `getCompleteDir` implementiert, die den kompletten Verbindungsaufbau und Einlog-Prozess übernimmt. Danach wechselt sie in das auszulesende Verzeichnis und liest alle Einträge ein.

Für jedes Unterverzeichnis wird dann die nicht-statische Variante von `getCompleteDir` aufgerufen.

```
bool CFtpConnection::getCompleteDir(const CUrl &url, CFileInfo &fi) {
  CBTRACELN("\ngetCompleteDir...\n-------------------------");

/*
** Auszulesendes Verzeichnis bestimmen
*/
  string serverPath="/"+url.getPath();
  if((fi.m_type==CFileInfo::INVALID)&&(serverPath.size()>1)) {
    string::size_type spos=serverPath.rfind("/");
    fi.m_path=serverPath.substr(0,spos);
    fi.m_name=serverPath.substr(spos+1);
  }

  CBTRACELN("Pfad: \""+fi.m_path+"\" Name: \""+fi.m_name+"\"");
  CBTRACELN("FTP-Pfad    : "+serverPath);

/*
** Port aus CUrl-Objekt extrahieren. Wenn keiner
** vorhanden, dann Port 21 nehmen
*/
  unsigned int port;
  if(url.getPort()!="")
    port=atol(url.getPort().c_str());
  else
    port=21;
```

Listing 451: Die statische Methode getCompleteDir

```
/*
** Usernamen aus CUrl-Objekt extrahieren. Ist keiner vorhanden,
** wird der in AnonymousName definierte verwendet.
*/
  string username=url.getUsername();
  if(username=="")
    username=AnonymousName;

/*
** Passwort aus CUrl-Objekt extrahieren. Ist keins vorhanden,
** wird das in AnonymousPwd definierte verwendet.
*/
  string password=url.getPassword();
  if(password=="")
    password=AnonymousPwd;

  try {

/*
** Verbindungsaufbau zu Server
*/
    CFtpConnection ftp(url.getDomain(),port);

/*
** Einloggen
*/
    if(!ftp.login(username, password)) {
      CBTRACELN("SgetCompleteDir: Login fehlgeschlagen");
      return(false);
    }

/*
** In das auszulesende Verzeichnis wechseln
*/
    if(!ftp.changeDir(serverPath)) {
      CBTRACELN("SgetCompleteDir: Verzeichniswechsel \
fehlgeschlagen");
      return(false);
    }

    if(fi.m_type==CFileInfo::INVALID)
```

Listing 451: Die statische Methode getCompleteDir (Forts.)

```
      fi.m_type=CFileInfo::DIR;

    fi.m_filelist=new CFileInfo::filelist_type;

/*
** Aktuelles  Verzeichnis einlesen
*/
    if(!ftp.list(*fi.m_filelist)) {
      CBTRACELN("SgetCompleteDir: list fehlgeschlagen");
      return(false);
    }

/*
** Einträge sortieren
*/
    sortFileList(*fi.m_filelist);

/*
** Alle Einträge durchlaufen und für Verzeichnisse
** die nicht-statische getCompleteDir-Methode aufrufen
*/
    for(CFileInfo::filelist_type::iterator iter=
                                fi.m_filelist->begin();
        iter!=fi.m_filelist->end();
        ++iter) {
      if(iter->m_type==CFileInfo::DIR) {
        CBTRACELN("Rekursiver Aufruf fuer: "+serverPath+"/"+
                  iter->m_name);
        ftp.getCompleteDir(serverPath+"/"+iter->m_name, *iter);
      }
    }

/*
** Für aktuelles Verzeichnis die Größe bestimmmen
*/
    fi.m_size=CFileSize();
    for(CFileInfo::filelist_type::iterator iter=
                                fi.m_filelist->begin();
        iter!=fi.m_filelist->end();
        ++iter)
      fi.m_size+=iter->m_size;
```

Listing 451: Die statische Methode getCompleteDir (Forts.)

```
  }
  catch(internet_error) {
    CBTRACELN("SgetCompleteDir fehlgeschlagen (Exception!)");
    return(false);
  }

  CBTRACELN("SgetCompleteDir Erfolg!");
  return(true);
}
```

Listing 451: Die statische Methode getCompleteDir (Forts.)

Die nun folgende nicht-statische Version von `getCompleteDir` baut auf der bereits erstellten Verbindung zum FTP-Server auf, liest die Verzeichnisse ein und ruft sich bei weiteren Verzeichnissen rekursiv auf.

```
bool CFtpConnection::getCompleteDir(const string &dir,
                                    CFileInfo &fi) {

  try {

/*
** In das auszulesende Unterverzeichnis wechseln
*/
    if(!changeDir(dir)) {
      CBTRACELN("getCompleteDir: Verzeichniswechsel \
fehlgeschlagen");
      return(false);
    }

    fi.m_filelist=new CFileInfo::filelist_type;

/*
** Aktuelles Verzeichnis einlesen
*/
    if(!list(*fi.m_filelist)) {
      CBTRACELN("getCompleteDir: list fehlgeschlagen");
      return(false);
    }
```

Listing 452: Die nicht-statische Methode getCompleteDir

```
/*
** Einträge sortieren
*/
    sortFileList(*fi.m_filelist);

/*
** Alle Einträge durchlaufen und für Verzeichnisse
** die nicht-statische getCompleteDir-Methode aufrufen
*/
    for(CFileInfo::filelist_type::iterator iter=
                                  fi.m_filelist->begin();
        iter!=fi.m_filelist->end();
        ++iter) {
      if(iter->m_type==CFileInfo::DIR) {
        CBTRACELN("Rekursiver Aufruf fuer: "+dir+"/"+iter->m_name);
        getCompleteDir(dir+"/"+iter->m_name, *iter);
      }
    }

/*
** Für aktuelles Verzeichnis die Größe bestimmmen
*/
    fi.m_size=CFileSize();
    for(CFileInfo::filelist_type::iterator iter=
                                  fi.m_filelist->begin();
        iter!=fi.m_filelist->end();
        ++iter)
      fi.m_size+=iter->m_size;

  }
  catch(internet_error) {
    CBTRACELN("getCompleteDir fehlgeschlagen (Exception!)");
    return(false);
  }

  CBTRACELN("getCompleteDir Erfolg!");
  return(true);
}
```

Listing 452: Die nicht-statische Methode getCompleteDir (Forts.)

Die Klasse `CFtpConnection` finden Sie auf der CD in den `CFtpConnection`-Dateien.

Dateiverwaltung

Dieses Kapitel beschäftigt sich mit der Dateiverwaltung in C++. Obwohl C++ mächtige Stream-Bibliotheken zum Lesen und Schreiben von Daten aus und in Dateien besitzt, werden grundlegende Operatoren des Datei-Systems wie z.B. das Anlegen oder Löschen von Verzeichnissen nur stiefmütterlich oder überhaupt nicht vom ANSI-Standard abgedeckt.

Wir werden daher an einigen Kernpunkten nicht daran vorbeikommen, plattformabhängige Funktionen einzusetzen.

Dieser Einsatz soll aber der besseren Portierung wegen auf ein Minimum begrenzt werden und betrifft lediglich die Rezepte 80 bis 85. Alle anderen Funktionen greifen – wenn notwendig – auf die in diesen sechs Rezepten definierten Funktionen zurück.

Um alle Rezepte dieses Kapitels anwenden zu können, müssen bei einer Portierung auf ein anderes System als Windows nur diese sechs Methoden an die neue Umgebung angepasst werden.

80 Wie wird eine Datei kopiert?

Die ANSI-Norm von C++ stellt keine direkte Möglichkeit zur Verfügung, eine Datei zu kopieren. Eine mögliche Lösung wäre das Kopieren über Datei-Ströme. (Die Daten eines Eingabe-Stroms werden in einen Ausgabe-Strom geschrieben.)

Dieser Ansatz ist im Allgemeinen erheblich zeitaufwändiger als die vom entsprechenden System bzw. Compiler zur Verfügung gestellte Kopierfunktion.

Deswegen setzen wir hier am Beispiel des Betriebssystems Windows eine solche Systemfunktion ein, indem wir sie in einer eigenen Funktion kapseln.

copyFile

`copyFile` besitzt folgende Parameter:

- `sname` – Name der Quelldatei
- `dname` – Name der Zieldatei
- `dontOverwrite` – boolescher Wert, der darüber entscheidet, ob eine bereits unter dem Zielnamen existierende Datei überschrieben wird oder nicht. (Standard: true)

Der Quellcode gestaltet sich folgendermaßen:

```
bool copyFile(const std::string &sname,
              const std::string &dname,
              bool dontOverwrite) {
  return(CopyFile(sname.c_str(), dname.c_str(), dontOverwrite)!=0);
}
```

Listing 453: Die Funktion copyFile

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien. Die Windows-Systemfunktion `CopyFile` ist in der Header-Datei *windows.h* definiert und auf gängigen Compilern (Microsoft, Borland etc.) verfügbar.

81 Wie wird eine Datei gelöscht?

Das Löschen von Dateien zählt zu den systemspezifischen Aufgaben. Wir setzen am Beispiel von Windows die in *windows.h* definierte Funktion `DeleteFile` ein und kapseln sie in unserer eigenen Funktion `removeFile`.

removeFile

```
bool removeFile(const string &name) {
  return(DeleteFile(name.c_str())!=0);
}
```

Listing 454: Die Funktion removeFile

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien.

82 Wie wird ein Verzeichnis erstellt?

Auch hier müssen wir auf eine Systemfunktion zurückgreifen. Für Windows existiert dazu die in *windows.h* definierte Funktion `CreateDirectory`, die wir in einer eigenen Funktion kapseln.

createDirectory

```
bool createDirectory(const std::string &name) {
  return(CreateDirectory(name.c_str(),NULL)!=0);
}
```

Listing 455: Die Funktion createDirectory

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien.

83 Wie wird ein Verzeichnis gelöscht?

Das Gegenstück zum Erstellen eines Verzeichnisses bedarf ebenfalls einer Systemfunktion. Sie heißt `RemoveDirectory`, ist in *windows.h* definiert und löscht ein leeres Verzeichnis. Wir kapseln sie in der Funktion `removeDirectoy`.

removeDirectory

```
bool removeDirectory(const string &name) {
  return(RemoveDirectory(name.c_str())!=0);
}
```

Listing 456: Die Funktion removeDirectory

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien.

84 Wie können Datei-Informationen ermittelt werden?

Um die später zu ermittelnden Datei-Informationen abspeichern zu können, wird eine Klasse `CFileInfo` implementiert.

CFileInfo

Sie besitzt folgende Attribute:

- `m_path` – der Pfad der Datei (nur Pfad ohne abschließendes `/`)
- `m_name` – der Name der Datei (nur Dateiname)
- `m_type` – der Dateityp. Es existieren die Werte INVALID für ungültige Datei (z.B. nicht gefunden), FILE für eine herkömmliche Datei und DIR für ein Verzeichnis.

- `m_size` – die Größe der Datei in Bytes
- `m_created` – Zeitpunkt, an dem die Datei erstellt wurde.
- `m_modified` – Zeitpunkt, an dem die Datei zuletzt verändert wurde, also Schreibzugriff stattgefunden hat.
- `m_filelist` – für den Fall eines Verzeichnisses kann dieser Container die im Verzeichnis enthaltenen Dateien aufnehmen. Auf diese Weise lassen sich später Verzeichnisbäume erstellen.

Klassendefinition

```
class CFileInfo {
public:
  typedef std::vector<CFileInfo> filelist_type;
  enum FTYPE {DIR=0, FILE, INVALID};

  std::string m_path;
  std::string m_name;
  FTYPE m_type;
  CFileSize m_size;
  CMoment m_created;
  CMoment m_modified;
  filelist_type *m_filelist;
};
```

Listing 457: Klassendefinition von CfileInfo

`CFileInfo` macht von der Klasse `CMoment` aus Rezept 61 Gebrauch.

Konstruktoren

Damit `CFileInfo` problemlos mit der STL zusammenarbeiten kann, bekommt die Klasse einen Standard- und einen Kopier-Konstruktor.

```
  CFileInfo()
  : m_type(INVALID), m_filelist(0)
  {}

  CFileInfo(const CFileInfo &fi)
  : m_filelist(0) {
```

Listing 458: Die Konstruktoren von CFileInfo

```
/*
** Aufruf des = Operators
*/
    *this=fi;
  }
```

Listing 458: Die Konstruktoren von CFileInfo (Forts.)

Destruktor

Damit keine Speicher-Lecks entstehen, gibt der Destruktor eine eventuell dynamisch angelegte Dateiliste wieder frei.

```
  ~CFileInfo() {
    if(m_filelist)
      delete(m_filelist);
  }
};
```

Listing 459: Die Destruktor von CFileInfo

Zuweisungsoperator

Ein eigener Zuweisungsoperator, der eine tiefe Objekt-Kopie anfertigt, wird hinzugefügt.

```
  CFileInfo &operator=(const CFileInfo &fi) {

/*
** Zuweisung an sich selbst?
** => Abbruch
*/
    if(this==&fi)
      return(*this);

/*
** Existiert alte Filelist?
** => Freigeben
*/
```

Listing 460: Der Zuweisungsoperator von CFileInfo

```
    if(m_filelist)
      delete(m_filelist);

/*
** Besitzt Quelle eine Filelist?
** => Neue filelist anlegen und kopieren
*/
    if(fi.m_filelist) {
      m_filelist=new filelist_type;
      *m_filelist=*fi.m_filelist;
    }
    else
      m_filelist=0;

/*
** Restliche Attribute kopieren
*/
    m_path=fi.m_path;
    m_name=fi.m_name;
    m_type=fi.m_type;
    m_size=fi.m_size;
    m_created=fi.m_created;
    m_modified=fi.m_modified;
    return(*this);
  }
```

Listing 460: Der Zuweisungsoperator von CFileInfo (Forts.)

getShallowCopy

Der Zuweisungsoperator fertigt eine tiefe Kopie des Objektes an, er kopiert also auch eine eventuell bei `m_filelist` vorhandene Unterverzeichnis-Struktur.

Es kann aber durchaus Sinn machen, von einem `CFileInfo`-Objekt nur eine flache Kopie anzufertigen (beispielsweise beim Suchen nach bestimmten Dateien.)

Aus diesem Grunde fügen wir der Klasse eine Methode `getShallowCopy` bei, die eine solche flache Kopie anfertigt:

```
  CFileInfo getShallowCopy() const {
    CFileInfo cpy;
```

Listing 461: Die Methode getShallowCopy von CFileInfo

```
    cpy.m_path=m_path;
    cpy.m_name=m_name;
    cpy.m_type=m_type;
    cpy.m_size=m_size;
    cpy.m_created=m_created;
    cpy.m_modified=m_modified;
    cpy.m_filelist=0;
    return(cpy);
  }
```

Listing 461: Die Methode getShallowCopy von CFileInfo (Forts.)

Vergleichsoperator

Vielleicht sollen Objekte vom Typ `CFileInfo` später sortiert werden, deswegen fügen wir noch den <-Operator hinzu.

```
  bool operator<(const CFileInfo &fi) const {

/*
** Verzeichnisse vor Dateien vor ungültigen Einträgen
*/
    if(m_type<fi.m_type)
      return(true);
    if(m_type>fi.m_type)
      return(false);

/*
** Namen vergleichen, Groß- und Kleinschreibung ignorieren
*/
    insensitive_string a=m_name.c_str();
    insensitive_string b=fi.m_name.c_str();
    if(a<b)
      return(true);
    if(a>b)
      return(false);

/*
** Wenn Namen gleich sind, dann zwischen Groß-
** und Kleinschreibung unterscheiden
*/
```

Listing 462: operator< von CFileInfo

```
    if(m_name<=fi.m_name)
      return(true);
    else
      return(false);
  }
```

Listing 462: operator< von CFileInfo (Forts.)

Der von der ANSI-Norm vorgeschriebene Wertebereich von `unsigned long` reicht von 0 bis 2^{32} und ist damit viel zu klein, um die Größen heute möglicher Dateien abzudecken. Deswegen werden zur Darstellung von Dateigrößen üblicherweise zwei `unsigned long`-Werte benutzt.

Um diese zwei Werte besser handhaben zu können, wurde die Klasse `CFileSize` implementiert, die Vergleichsoperatoren und einen Additionsoperator besitzt, um später verschiedene Dateigrößen bequem aufaddieren zu können.

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`.

Es muss darauf geachtet werden, dass die Struktur verschachtelt sein kann: Sollte `m_filelist` auf eine weitere `CFileInfo`-Struktur zeigen, so muss diese auch gespeichert werden.

Um dies beim Laden zu berücksichtigen, wird aus technischen Gründen ein boolescher Wert abgespeichert, der mitteilt, ob eine Unterstruktur zu laden ist oder nicht.

```
friend CBinaryOStream &operator<<(CBinaryOStream &os,
                                  const CFileInfo &fi){
  os << fi.m_path;
  os << fi.m_name;
  os.write(&fi.m_type, sizeof(fi.m_type));
  os << fi.m_size;
  os << fi.m_created;
  os << fi.m_modified;
  bool subdir=(fi.m_filelist!=0);
  os << subdir;
  if(subdir)
```

Listing 463: Die Stream-Operatoren von CFileInfo

```
    os << *fi.m_filelist;
  return(os);
}

//*********

friend CBinaryIStream &operator>>(CBinaryIStream &is,
                                  CFileInfo &fi){
  is >> fi.m_path;
  is >> fi.m_name;
  is.read(&fi.m_type, sizeof(fi.m_type));
  is >> fi.m_size;
  is >> fi.m_created;
  is >> fi.m_modified;
  if(fi.m_filelist) {
    delete(fi.m_filelist);
    fi.m_filelist=0;
  }
  bool subdir;
  is >> subdir;
  if(subdir) {
    fi.m_filelist=new CFileInfo::filelist_type;
    is >> *fi.m_filelist;
  }
  return(is);
}
```

Listing 463: Die Stream-Operatoren von CFileInfo (Forts.)

CFileSize

Die Klasse `CFileSize` wird implementiert, um die 64 Bit große Dateigröße einfacher verwalten und bearbeiten zu können.

Klassendefinition

```
class CFileSize {
public:
  unsigned long m_lowsize;
  unsigned long m_highsize;
};
```

Listing 464: Die Klassendefinition von CFileSize

Konstruktoren

Weil Zahlen der Größe von `CFileSize` nicht ohne weiteres dargestellt werden können, wurde ein Konstruktor implementiert, der ein `CFileSize`-Objekt aus einem `CHugeNumber`-Objekt aus Rezept 119 erzeugt.

```
  CFileSize(unsigned long low, unsigned long high)
    : m_lowsize(low), m_highsize(high)
  {}

/*
** Standard-Konstruktor
*/
  CFileSize()
    : m_lowsize(0), m_highsize(0)
  {}

/*
** Kopier-Konstruktor
*/
  CFileSize(const CFileSize &fs)
    : m_lowsize(fs.m_lowsize), m_highsize(fs.m_highsize)
  {}

  CFileSize(const CHugeNumber &n) {
    *this=n.getAsFileSize();
  }
```

Listing 465: Die Konstruktoren von CFileSize

Vergleichsoperatoren

```
  bool operator<(const CFileSize &fs) const {
    if (m_highsize<fs.m_highsize)
      return(true);
    if ((m_highsize==fs.m_highsize)&&
      (m_lowsize<fs.m_lowsize))
      return(true);
    return(false);
  }
```

Listing 466: Die Vergleichsoperatoren von CFileSize

```
//*********

  bool operator==(const CFileSize &fs) const {
    return((m_lowsize==fs.m_lowsize)&&
           (m_highsize==fs.m_highsize));
  }
```

Listing 466: Die Vergleichsoperatoren von CFileSize (Forts.)

Rechenoperatoren

```
  CFileSize operator+(const CFileSize &fs) const {
    CFileSize tmp;
    tmp.m_highsize=m_highsize+fs.m_highsize;
    tmp.m_lowsize=m_lowsize+fs.m_lowsize;
    if(tmp.m_lowsize<m_lowsize)
      tmp.m_highsize++;
    return(tmp);
  }

//*********

  CFileSize &operator+=(const CFileSize &fs) {
    m_highsize+=fs.m_highsize;
    if((m_lowsize+fs.m_lowsize)<m_lowsize)
      m_highsize++;
    m_lowsize+=fs.m_lowsize;
    return(*this);
  }
};
```

Listing 467: Die Rechenoperatoren von CFileSize

Stream-Operatoren

Als Bestandteil von `CFileInfo` muss `CFileSize` auch mit den Stream-Operatoren arbeiten können, wir implementieren daher noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`.

```
friend CBinaryOStream &operator<<(CBinaryOStream &os,
                                  const CFileSize &fs){
  os << fs.m_lowsize;
  os << fs.m_highsize;
  return(os);
}

//*********

friend CBinaryIStream &operator>>(CBinaryIStream &is,
                                  CFileSize &fs){
  is >> fs.m_lowsize;
  is >> fs.m_highsize;
  return(is);
}
```

Listing 468: Die Stream-Operatoren für CFileSize

Die Klassen `CFileInfo` und `CFileSize` finden Sie auf der CD in der *CFileInfo.h*-Datei.

getFileInfo

Die Funktion `getFileInfo` schließlich bekommt einen Dateinamen übergeben, erstellt für diese Datei ein `CFileInfo`-Objekt und liefert es zurück.

Sie macht von der Windows-spezifischen Funktion `FindFirstFile` und der Struktur `WIN32_FIND_DATA` Gebrauch, die in *windows.h* definiert sind.

```
CFileInfo getFileInfo(const string &name) {
  CFileInfo fe;

/*
** Windows-spezifische Struktur für Datei-Infos
*/
  WIN32_FIND_DATA data;

/*
** Konnte Datei nicht geöffnet werden?
** => Leere (auf INVALID gesetzte) FileInfo zurückgeben
*/
  HANDLE hdl=FindFirstFile(name.c_str(), &data);
```

Listing 469: Die Funktion getFileInfo

```
  if(hdl==INVALID_HANDLE_VALUE)
    return(fe);

/*
** Erstellungsdatum ermitteln
*/
  SYSTEMTIME stime;
  FileTimeToSystemTime(&data.ftCreationTime, &stime);
  fe.m_created=CMoment(stime.wYear, stime.wMonth, stime.wDay,
    stime.wHour, stime.wMinute, stime.wSecond, stime.wMilliseconds);

/*
** Datum der letzten Änderung ermitteln
*/
  FileTimeToSystemTime(&data.ftLastWriteTime, &stime);
  fe.m_modified=CMoment(stime.wYear, stime.wMonth, stime.wDay,
    stime.wHour, stime.wMinute, stime.wSecond, stime.wMilliseconds);

/*
** Dateiname und Pfad ermitteln
*/
  fe.m_name=data.cFileName;
  if(name.find("/")!=string::npos)
    fe.m_path=name.substr(0,name.rfind("/"));
  else
    fe.m_path="";

/*
** Dateigröße ermitteln
*/
  fe.m_size=CFileSize(data.nFileSizeLow,data.nFileSizeHigh);

/*
** Typ bestimmen
*/
  if(data.dwFileAttributes&FILE_ATTRIBUTE_DIRECTORY)
    fe.m_type=CFileInfo::DIR;
  else
    fe.m_type=CFileInfo::FILE;

  FindClose(hdl);
```

Listing 469: Die Funktion getFileInfo (Forts.)

```
  return(fe);
}
```

Listing 469: Die Funktion getFileInfo (Forts.)

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien.

85 Wie kann ein Verzeichnis ausgelesen werden?

Die Methode `getDir` bekommt den Namen eines Verzeichnisses und eine Referenz auf einen Container übergeben. Sie ermittelt dann alle Dateien und Unterverzeichnisse, speichert sie im Container und liefert Erfolg oder Misserfolg als booleschen Wert zurück.

Sie speichert die Datei-Informationen als `CFileInfo`-Objekte. Die Klasse wurde in Rezept 84 eingeführt.

Sie macht von den Windows-spezifischen Funktionen `FindFirstFile` und `FindNextFile` und der Struktur `WIN32_FIND_DATA` Gebrauch, die in `windows.h` definiert sind.

Da unter Windows bei dem Durchlaufen eines Verzeichnisses die Datei-Informationen automatisch verfügbar sind, wäre es Verschwendung von Laufzeit, die Datei-Informationen über `getFileInfo` aus Rezept 84 zu ermitteln.

Bei anderen Systemen könnte es aber durchaus Sinn machen, auf diese Funktion zurückzugreifen.

getDir

```
bool getDir(const std::string &name,
            CFileInfo::filelist_type &files) {

/*
** Windows-spezifische Struktur für Datei-Infos
*/
  WIN32_FIND_DATA data;

/*
** Erste Datei des Verzeichnisses ermitteln
*/
  HANDLE hdl=FindFirstFile((name+"/*").c_str(), &data);
```

Listing 470: Die Funktion getDir

```
/*
** Konnte nicht im Verzeichnis gelesen werden?
** => Abbruch
*/
  if(hdl==INVALID_HANDLE_VALUE)
    return(false);
  do {
    CFileInfo fe;

/*
** Erstellungsdatum ermitteln
*/
    SYSTEMTIME stime;
    FileTimeToSystemTime(&data.ftCreationTime, &stime);
    fe.m_created=CMoment(stime.wYear, stime.wMonth, stime.wDay,
      stime.wHour, stime.wMinute, stime.wSecond,
      stime.wMilliseconds);

/*
** Datum der letzten Änderung ermitteln
*/
    FileTimeToSystemTime(&data.ftLastWriteTime, &stime);
    fe.m_modified=CMoment(stime.wYear, stime.wMonth, stime.wDay,
      stime.wHour, stime.wMinute, stime.wSecond,
      stime.wMilliseconds);

/*
** Dateiname und Pfad ermitteln
*/
    fe.m_name=data.cFileName;
    fe.m_path=name;

/*
** Dateigröße ermitteln
*/
    fe.m_size=CFileSize(data.nFileSizeLow,data.nFileSizeHigh);

/*
** Typ bestimmen
*/
```

Listing 470: Die Funktion getDir (Forts.)

```
    if(data.dwFileAttributes&FILE_ATTRIBUTE_DIRECTORY)
      fe.m_type=CFileInfo::DIR;
    else
      fe.m_type=CFileInfo::FILE;

/*
** Kein Standardverzeichnis? (wie . oder ..)
** => Zur Liste hinzufügen
*/
    if((fe.m_name!=".")&&(fe.m_name!=".."))
      files.push_back(fe);

/*
** Nächste Datei ermitteln und bei Misserfolg
** die Schleife abbrechen
*/
  } while(FindNextFile(hdl, &data));

  FindClose(hdl);
  return(true);
}
```

Listing 470: Die Funktion getDir (Forts.)

Die Funktion finden Sie auf der CD in den `FileSpecifics`-Dateien.

Das folgende Beispiel fragt nach einem Verzeichnis-Namen und gibt dann dessen Einträge aus:

```
#include "FileSpecifics.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  cout << "Verzeichnis:";
  string dir;
  getString(dir,200);
  CFileInfo::filelist_type direntries;
  if(getDir(dir,direntries)) {
```

Listing 471: Ein Beispiel zu getDir

```
    for(CFileInfo::filelist_type::iterator iter=direntries.begin();
        iter!=direntries.end();
        ++iter) {
      if(iter->m_type==CFileInfo::DIR)
        cout << "[DIR] ";
      cout << iter->m_name << endl;
    }
  }
  else
    cout << "ungueltiges Verzeichnis!" << endl;
}
```

Listing 471: Ein Beispiel zu getDir (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0606.cpp*.

86 Wie kann geprüft werden, ob eine Datei existiert?

Diese Aufgabe wird gewissermaßen bereits von der Funktion `getFileInfo` aus Rezept 84 erledigt. Denn sollte der Typ im zurück gelieferten `CFileInfo`-Objekt INVALID sein, dann existiert die Datei nicht.

fileExists

Das macht sich die Methode `fileExists` zu Nutze. Sie bekommt einen Namen übergeben und liefert über einen booleschen Wert die Information zurück, ob unter dem Namen eine Datei existiert.

```
bool fileExists(const std::string &name) {
  return(getFileInfo(name).m_type!=CFileInfo::INVALID);
}
```

Listing 472: Die Funktion fileExists

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

87 Wie kann eine Datei verschoben werden?

Obwohl es im Normalfall auch eine Systemfunktion zum Verschieben von Dateien gibt, können wir eine Datei auch mit unseren bisher geschriebenen Funktionen verschieben.

Wir kopieren die Datei zunächst mit `copyFile` aus Rezept 80 und löschen das Original anschließend mit `removeFile` aus Rezept 81

moveFile

`moveFile` besitzt folgende Parameter:

- `sname` – Name der Quelldatei
- `dname` – Name der Zieldatei
- `dontOverwrite` – boolescher Wert, der darüber entscheidet, ob eine bereits unter dem Zielnamen existierende Datei überschrieben wird oder nicht (Standard: true)

```
bool moveFile(const std::string &sname,
              const std::string &dname,
              bool dontOverwrite) {
  if(copyFile(sname.c_str(), dname.c_str(), dontOverwrite))
    return(removeFile(sname));
  else
    return(false);
}
```

Listing 473: Die Funktion moveFile

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

88 Wie wird ein Pfad erstellt?

Die Funktion `createDirectory` besitzt die Prämisse, dass das Verzeichnis, in dem das neue Verzeichnis erstellt werden soll, bereits existiert.

Damit ein Aufruf wie beispielsweise `createDirectory("c:/dira/dirb/dirc")` von Erfolg gekrönt ist, muss das Verzeichnis `c:/dira/dirb` bereits existieren.

Was aber, wenn nur `dira` oder keines der Verzeichnisse im Pfad bereits existiert? `createDirectory` würde scheitern.

createPath

Um Abhilfe zu schaffen, schreiben wir eine Funktion `createPath`, die nötigenfalls alle für den Pfad notwendigen Verzeichnisse erstellt. Der rekursive Ansatz greift auf `createDirectory` aus Rezept 82 zurück.

Da letzten Endes ein Verzeichnis erstellt werden soll, darf der Pfad nicht mit einem / (oder \ auf anderen Systemen) enden.[1]

```
bool createPath(const string &name) {

/*
** Verzeichnis erfolgreich erstellt?
** => Fertig
*/
  if(createDirectory(name))
    return(true);
  else {

/*
** Lässt sich Pfad weiter zerlegen?
** => createPath rekursiv aufrufen
*/
    if(name.rfind("/")!=string::npos) {
      createPath(name.substr(0,name.rfind("/")));
      return(createDirectory(name));
    }
    else
      return(false);
  }
}
```

Listing 474: Die Funktion createPath

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

89 Wie wird ein komplettes Verzeichnis kopiert?

Um ein komplettes Verzeichnis zu kopieren, müssen Sie alle Dateien des Verzeichnisses und alle Unterverzeichnisse kopieren. Ein rekursiver Ansatz bietet sich daher an.

1. Eine entsprechende Abfrage oder Anweisung zur Behebung dieser Einschränkung kann der Leser problemlos ergänzen.

copyDirectory

`copyDirectory` besitzt folgende Parameter:

- `sname` – Name der Quelldatei
- `dname` – Name der Zieldatei
- `copyMainDir` – boolescher Wert, der darüber entscheidet, ob das gesamte Verzeichnis kopiert werden soll (true) oder nur der Verzeichnisinhalt (false) (Standard: true)
- `dontOverwrite` – boolescher Wert, der darüber entscheidet, ob eine bereits unter dem Zielnamen existierende Datei überschrieben wird oder nicht (Standard: true)

Die Funktion macht Gebrauch von `createPath` aus Rezept 1, von `getDir` aus Rezept 85 und von `copyFile` aus Rezept 80.

```
bool copyDirectory(const string &sname,
                   const string &dname,
                   bool copyMainDir,
                   bool dontOverwrite) {

  string targetpath=dname;

/*
** Soll ganzes Verzeichnis kopiert werden?
** => Verzeichnis zu Zielpfad hinzufügen
*/
  if(copyMainDir)
    targetpath+="/"+sname.substr(sname.rfind("/")+1);

  bool success=true;
  CFileInfo::filelist_type files;

/*
** Zielpfad des aktuellen Verzeichnisses anlegen
*/
  createPath(targetpath);

/*
** Inhalt des Verzeichnisses ermitteln
```

Listing 475: Die Funktion copyDirectory

```
*/
  if(getDir(sname,files)) {

/*
** Alle Einträge des Verzeichnisses bearbeiten
*/
    for(CFileInfo::filelist_type::iterator iter=files.begin();
        iter!=files.end();
        ++iter){
      switch(iter->m_type) {

/*
** Unterverzeichnis?
** => Durch rekursiven Aufruf von copyDirectory kopieren
*/
        case CFileInfo::DIR:
          if((!(copyDirectory(sname+"/"+iter->m_name,
                              targetpath+"/"+iter->m_name,
                              false,
                              dontOverwrite)))&&(dontOverwrite))
            success=false;
          break;

/*
** Datei?
** => Mit copyFile kopieren
*/
        case CFileInfo::FILE:
          if((!(copyFile(sname+"/"+iter->m_name,
                         targetpath+"/"+iter->m_name,
                         dontOverwrite)))&&(dontOverwrite))
            success=false;
          break;
      }
    }
  }
  return(success);
}
```

Listing 475: Die Funktion copyDirectory (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

90 Wie kann ein komplettes Verzeichnis gelöscht werden?

Auch hier ist der einfachste Ansatz rekursiv, weil ein Verzeichnis Unterverzeichnisse besitzen kann, die gelöscht werden müssen, bevor das Wurzelverzeichnis gelöscht wird.

removeDirectory

`removeDirectory` besitzt folgende Parameter:

- `name` – Name des zu löschenden Verzeichnisses
- `removeMainDir` – boolescher Wert, der darüber entscheidet, ob das gesamte Verzeichnis gelöscht werden soll (true) oder nur der Verzeichnisinhalt (false)

Die Funktion macht von `getDir` aus Rezept 85, von `removeFile` aus Rezept 81 und von `removeDirectory` aus Rezept 83 Gebrauch.

```
bool removeDirectory(const string &name, bool removeMainDir) {

  bool success=true;
  CFileInfo::filelist_type files;

/*
** Inhalt des Verzeichnisses ermitteln
*/
  if(getDir(name,files)) {

/*
** Alle Einträge des Verzeichnisses bearbeiten
*/
    for(CFileInfo::filelist_type::iterator iter=files.begin();
        iter!=files.end();
        ++iter){
      switch(iter->m_type) {

/*
** Unterverzeichnis?
** => Durch rekursiven Aufruf von removeDirectory löschen
*/
        case CFileInfo::DIR:
```

Listing 476: Die Funktion removeDirectory

```
          if(!(removeDirectory(name+"/"+iter->m_name, true)))
            success=false;
          break;

/*
** Datei?
** => Mit removeFile kopieren
*/
        case CFileInfo::FILE:
          if(!(removeFile(name+"/"+iter->m_name)))
            success=false;
          break;
      }
    }
  }

/*
** Soll Wurzelverzeichnis ebenfalls gelöscht werden?
** => Verzeichnis löschen mit removeDirectory
*/
  if(removeMainDir)
    if(!(removeDirectory(name)))
      success=false;
  return(success);
}
```

Listing 476: Die Funktion removeDirectory (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

91 Wie kann ein komplettes Verzeichnis verschoben werden?

Anstatt eine komplett neue Funktion zum Verschieben eines Verzeichnisses zu implementieren, reduzieren wir das Verschieben auf ein Kopieren mit anschließendem Löschen der Originaldaten.

Auf diese Weise ist sichergestellt, dass die Originaldaten erst gelöscht werden, wenn die Kopier-Operation erfolgreich war.

Darüber hinaus ist es mit diesem Ansatz möglich, Ziel und Quelle an beliebigen Stellen zu wählen.

moveDirectory

`moveDirectory` besitzt folgende Parameter:

- `sname` – Name der Quelldatei
- `dname` – Name der Zieldatei
- `moveMainDir` – boolescher Wert, der darüber entscheidet, ob das gesamte Verzeichnis verschoben werden soll (true) oder nur der Verzeichnisinhalt (false) (Standard: true)
- `dontOverwrite` – boolescher Wert, der darüber entscheidet, ob eine bereits unter dem Zielnamen existierende Datei überschrieben wird oder nicht (Standard: true)

Die Funktion macht von `copyDirectory` aus Rezept 89 und von `removeDirectory` aus Rezept 90 Gebrauch.

```
bool moveDirectory(const string &sname,
                   const string &dname,
                   bool moveMainDir,
                   bool dontOverwrite) {
  if(copyDirectory(sname, dname, moveMainDir, dontOverwrite))
    return(removeDirectory(sname, moveMainDir));
  else
    return(false);
}
```

Listing 477: Die Funktion moveDirectory

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

92 Wie kann ein Verzeichnis-Inhalt sortiert werden?

Wenn Sie einmal einen von `getDir` erzeugten Container ausgeben, dann werden Sie feststellen, dass die Einträge nicht sortiert vorliegen.

Um diese Einträge einer Sortierung zu unterziehen, die dem gewohnten Bild entspricht, schreiben wir die Funktion `sortFileList`.

sortFileList

`sortFileList` verwendet den Quicksort-Algorithmus `sort` aus der Header-Datei `algorithm` sowie das Prädikat `less` aus der Header-Datei `functional`:

```
void sortFileList(CFileInfo::filelist_type &fl) {
  sort(fl.begin(), fl.end(), less<CFileInfo>());
}
```

Listing 478: Die Funktion sortFileList

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

Damit diese Funktion einwandfrei funktioniert, benötigt die Klasse `CFileInfo` einen <-Vergleichsoperator.

operator< für CFileInfo

Die `operator`<-Funktion unterscheidet zunächst zwischen dem Eintragstyp, denn Verzeichnisnamen sollen grundsätzlich vor Dateinamen stehen.

Ist der Eintragstyp gleich, dann werden die Namen verglichen, wobei nicht zwischen Groß- und Kleinschreibung unterschieden wird. Dazu setzen wir `insensitive_string` aus Rezept 17 ein.

Wenn die Namen identisch sind, können sie sich nur noch in der Groß- und Kleinschreibung unterscheiden (was unter Windows nie vorkommen kann). Deswegen werden die Namen dann noch einmal unter Berücksichtigung der Groß- und Kleinschreibung verglichen.

```
  bool operator<(const CFileInfo &fi) const {

/*
** Verzeichnisse vor Dateien vor ungültigen Einträgen
*/
    if(m_type<fi.m_type)
      return(true);
    if(m_type>fi.m_type)
      return(false);

/*
** Namen vergleichen, Groß- und Kleinschreibung ignorieren
*/
    insensitive_string a=m_name.c_str();
    insensitive_string b=fi.m_name.c_str();
    if(a<b)
```

Listing 479: operator< für CFileInfo

```
      return(true);
    if(a>b)
      return(false);

/*
** Wenn Namen gleich sind, dann zwischen Groß-
** und Kleinschreibung unterscheiden
*/
    if(m_name<=fi.m_name)
      return(true);
    else
      return(false);
  }
```

Listing 479: operator< für CFileInfo (Forts.)

Die Methode finden Sie in der Klasse `CFileInfo` auf der CD in der `FCileInfo.h`-Datei.

Das nachstehende Beispiel gibt ein vom Benutzer eingegebenes Verzeichnis sortiert aus:

```
#include "FileFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  cout << "Verzeichnis:";
  string dir;
  getString(dir,200);
  CFileInfo::filelist_type direntries;
  if(getDir(dir,direntries)) {
    sortFileList(direntries);
    for(CFileInfo::filelist_type::iterator iter=direntries.begin();
        iter!=direntries.end();
        ++iter) {
      if(iter->m_type==CFileInfo::DIR)
        cout << "[DIR] ";
      cout << iter->m_name << endl;
    }
```

Listing 480: Ein Beispiel zu sortFileList

```
  }
  else
    cout << "ungueltiges Verzeichnis!" << endl;
}
```

Listing 480: Ein Beispiel zu sortFileList (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0613.cpp*.

93 Wie können Informationen über ein komplettes Verzeichnis ermittelt werden?

Die Methode `getCompleteDir` funktioniert ähnlich wie `getDir`. Sie bekommt den Namen eines Verzeichnisses und eine Referenz auf einen Container übergeben, ermittelt dann durch rekursive Aufrufe alle Dateien und Unterverzeichnisse und speichert sie im `m_filelist`-Attribut. Sie liefert Erfolg oder Misserfolg als booleschen Wert zurück.

getCompleteDir

```
bool getCompleteDir(const string &name, CFileInfo &fi) {

/*
** Datei-Infos noch nicht geholt?
** => Dann jetzt
*/
  if(fi.m_type==CFileInfo::INVALID) {
    fi=getFileInfo(name);

/*
** Immer noch invalid?
** => Abbruch
*/
    if(fi.m_type==CFileInfo::INVALID)
      return(false);
  }

/*
** Gehört aktuelle Datei-Info zu einem Verzeichnis?
```

Listing 481: Die Funktion getCompleteDir

```
*/
  if(fi.m_type==CFileInfo::DIR) {

/*
** Neue Datei-Liste anlegen
*/
    fi.m_filelist=new CFileInfo::filelist_type;

/*
** Liste füllen und sortieren
*/
    getDir(name, *fi.m_filelist);
    sortFileList(*fi.m_filelist);

/*
** Für Unterverzeichnisse getCompleteDir rekursiv aufrufen
*/
    CFileInfo::filelist_type::iterator iter=fi.m_filelist->begin();
    while((iter!=fi.m_filelist->end())&&
          (iter->m_type==CFileInfo::DIR)) {
      getCompleteDir(name+"/"+iter->m_name, *iter);
      ++iter;
    }
  return(true);
}
```

Listing 481: Die Funktion getCompleteDir (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

94 Wie kann die tatsächliche Größe eines Verzeichnisses (mitsamt Unterverzeichnissen) ermittelt werden?

Um dieses Problem zu lösen, ergänzen wir die Funktion `getCompleteDir` aus Rezept 93 um einige Anweisungen.

Nachdem bei einem Verzeichnis alle Unterverzeichnisse und Dateien in `m_filelist` gespeichert sind, addiert eine Schleife alle Einzelgrößen zusammen und speichert die daraus resultierende Gesamtgröße in `m_size` des Verzeichnisses.

getCompleteDir

```
bool getCompleteDir(const string &name, CFileInfo &fi) {

/*
** Datei-Infos noch nicht geholt?
** => Dann jetzt
*/
  if(fi.m_type==CFileInfo::INVALID) {
    fi=getFileInfo(name);

/*
** Immer noch invalid?
** => Abbruch
*/
    if(fi.m_type==CFileInfo::INVALID)
      return(false);
  }

/*
** Gehört aktuelle Datei-Info zu einem Verzeichnis?
*/
  if(fi.m_type==CFileInfo::DIR) {

/*
** Neue Datei-Liste anlegen
*/
    fi.m_filelist=new CFileInfo::filelist_type;

/*
** Liste füllen und sortieren
*/
    getDir(name, *fi.m_filelist);
    sortFileList(*fi.m_filelist);

/*
** Für Unterverzeichnisse getCompleteDir rekursiv aufrufen
*/
    CFileInfo::filelist_type::iterator iter=fi.m_filelist->begin();
    while((iter!=fi.m_filelist->end())&&
          (iter->m_type==CFileInfo::DIR)) {
```

Listing 482: Die erweiterte getCompleteDir-Funktion

```
      getCompleteDir(name+"/"+iter->m_name, *iter);
      ++iter;
    }

/*
** Datei-Größe des Verzeichnisses auf 0 setzen
*/
    fi.m_size=CFileSize();

/*
** Größen aller Unterverzeichnisse und Dateien aufaddieren
*/
    for(iter=fi.m_filelist->begin();
        iter!=fi.m_filelist->end();
        ++iter)
      fi.m_size+=iter->m_size;
  }
  return(true);
}
```

Listing 482: Die erweiterte getCompleteDir-Funktion (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

95 Wie kann ein Verzeichnis dargestellt werden?

Wie bereits in vorangegangenen Kapiteln wollen wir zur Darstellung HTML einsetzen. Leicht angelehnt an eine typische Browser-Darstellung wollen wir eine Funktion `dirToHtml` schreiben, die eine Ausgabe wie in Abbildung 38 erzeugt.

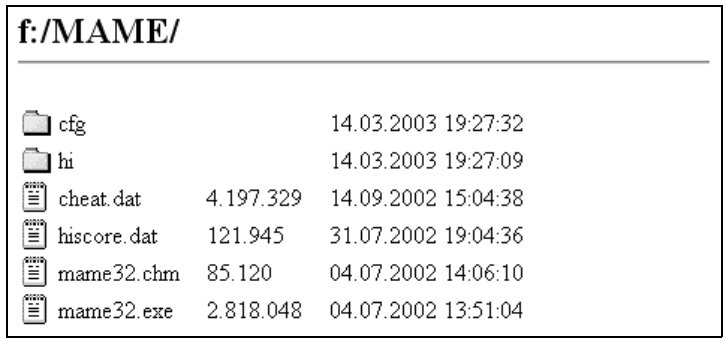


Abbildung 38: Ein Verzeichnis in HTML

Um den HTML-String in eine Datei zu schreiben, kann die Funktion `saveString-AsHtml` aus Rezept 57 eingesetzt werden.

dirToHtml

Die Funktion existiert in zwei Varianten. In der ersten existieren die auszugebenden Einträge bereits in einer `CFileInfo::filelist_type`-Struktur. Der Verzeichnisname wird nur noch angegeben, damit er von `dirToHtml` ausgegeben werden kann:

```
string dirToHtml(const CFileInfo::filelist_type &filelist,
                 const string &dirname) {

/*
** Pfad als Überschrift mit Linie schreiben
*/
  string str;
  str+="<h2>"+dirname+"/<hr /></h2>\n";

/*
** Tabelle erzeugen und alle Einträge durchlaufen
*/
  str+="<table>\n";

  for(CFileInfo::filelist_type::const_iterator iter=
                                      filelist.begin();
      iter!=filelist.end();
      ++iter) {
    str+="<tr><td>";

/*
** Entsprechend des Eintrag-Typs die Grafik
** t_dir.gif oder t_file.gif verwenden
*/
    switch(iter->m_type) {
      case CFileInfo::DIR:
        str+="<img src=\"t_dir.gif\" width=\"20\" />";
        break;

      case CFileInfo::FILE:
        str+="<img src=\"t_file.gif\" width=\"20\" />";
        break;
    }
```

Listing 483: Die funktion dirToHtml für filelist_type

```
/*
** Name des Eintrags schreiben
*/
    str+="</td><td>";
    str+=iter->m_name;
    str+="</td><td width=\"10\">";

/*
** Eintrag vom Typ FILE?
** => Dateigröße schreiben
*/
    if(iter->m_type==CFileInfo::FILE) {
      str+="</td><td>";
      str+=CHugeNumber(iter->m_size.m_lowsize,
           iter->m_size.m_highsize).getAsPointedNumber();
    }
    else
      str+="</td><td>";

/*
** Datum der letzten Änderung schreiben
*/
    str+="</td><td width=\"10\"></td><td>";
    str+=iter->m_modified.getDDMMYYYYHHMMSS();

/*
** Tabellenzelle und -zeile beenden
*/
    str+="</td></tr>\n";
  }

/*
** Tabelle beenden und String zurückliefern
*/
  str+="</table>\n";
  return(str);
}
```

Listing 483: Die funktion dirToHtml für filelist_type (Forts.)

Die zweite Variante liest das Verzeichnis eigenständig ein (mit Hilfe von `getDir` aus Rezept 85), sortiert die Einträge mit `sortFileList` aus Rezept 92 und ruft dann die erste Variante der `dirToHtml`-Funktion auf.

```
string dirToHtml(const string &name) {

/*
** Informationen über darzustellendes
** Verzeichnis einholen
*/
  CFileInfo dirinfo=getFileInfo(name);

/*
** Kein Verzeichnis?
** => Abbruch
*/
  if(dirinfo.m_type!=CFileInfo::DIR)
    return("");

/*
** Verzeichnis-Inhalt ermitteln und sortieren
*/
  CFileInfo::filelist_type fl;
  getDir(name,fl);
  sortFileList(fl);
  return(dirToHtml(fl,dirinfo.m_path));
}
```

Listing 484: dirToHtml mit Verzeichnisnamen als Parameter

Die Funktionen finden Sie auf der CD in den `FileFunctions`-Dateien.

Zur Darstellung der Dateigröße wird die Klasse `CHugeNumber` aus Rezept 119 verwendet.

Damit die verwendeten Grafiken korrekt dargestellt werden, müssen sich im selben Verzeichnis mit dem erzeugten HTML-Dokument die Grafiken `t_dir.gif` und `f_file.gif` aus dem `images`-Ordner auf der CD befinden.

96 Wie kann ein Verzeichnisbaum dargestellt werden?

Wir haben in Rezept 94 die Funktion `getCompleteDir` geschrieben, die einen kompletten Verzeichnisbaum in einer eigenen Datenstruktur abbildet und für jedes Verzeichnis die tatsächliche Größe bestimmt.

Es wäre nun auch schön, diesen Verzeichnisbaum grafisch darstellen zu können. Wir schreiben dazu die Funktionen `fileInfoToHtml`, die den HTML-Code erzeugen, um den Verzeichnisbaum wie in Abbildung 39 gezeigt darzustellen.

Eventuelle Schönheitskorrekturen können problemlos vom Leser vorgenommen werden.

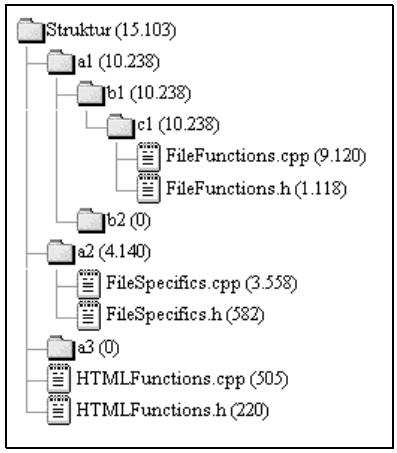


Abbildung 39: Ein Verzeichnisbaum in HTML

fileInfoToHtml I

Um die Darstellung ein wenig flexibel zu halten, besitzt die Funktion diverse Parameter[2].

- `fi` – eine Referenz auf den Verzeichnisbaum
- `onlyDirs` – ein boolescher Wert, der entscheidet, ob nur Verzeichnisse (true) oder auch Dateien (false) dargestellt werden sollen
- `imgsize` – Größe der Grafiken
- `fontsize` – Größe des Textes

2. Je nach Bedarf kann der Leser diese Parameter erweitern oder deren Bedeutung ändern.

Der Verzeichnisbaum in Abbildung 39 wurde beispielsweise mit `imgsize=20` und `fontsize=2` erzeugt.


```
string fileInfoToHtml(const CFileInfo &fi,
                      bool onlyDirs,
                      int imgsize,
                      int fontsize) {
  string str;

/*
** Schriftgröße setzen
*/
  str+="<font size=\""+toString(fontsize)+"\">\n";

/*
** Grafik für das Hauptverzeichnis mitsamt
** Namen und Dateigröße setzen
*/
  str+="<img src=\"t_dir.gif\" align=\"middle\" width=\""+
       toString(imgsize)+"\" />";
  str+=fi.m_name+" ("+
       CHugeNumber(fi.m_size.m_lowsize,
                   fi.m_size.m_highsize).getAsPointedNumber()+
       ")";
  str+="<br />\n";

/*
** Besitzt Hauptverzeichnis Dateien oder Unterverzeichnisse?
** => Überladenes fileInfoToHtml für jeden Eintrag aufrufen
*/
  if(fi.m_filelist&&(fi.m_filelist->size()>0))
    for(unsigned int i=0; i<fi.m_filelist->size(); i++)

/*
** Um korrekten grafischen Ast darstellen zu können,
** wird übergeben, ob es der letzte Eintrag ist oder nicht
*/
      str+=fileInfoToHtml((*fi.m_filelist)[i],
                          "",
                          onlyDirs,
                          imgsize,
```

Listing 485: Die Funktion fileInfoToHtml

```
                          (i==(fi.m_filelist->size()-1)));

  str+="</font>\n";
  return(str);
}
```

Listing 485: Die Funktion fileInfoToHtml (Forts.)

Die Funktion ruft rekursiv eine überladene Variante von sich auf.

fileInfoToHtml II

```
string fileInfoToHtml(const CFileInfo &fi,
                      string line,
                      bool onlyDirs,
                      int imgsize,
                      bool last) {

/*
** Eintrag ist Datei, aber es sollen nur Verzeichnisse dargestellt werden?
** => Abbruch
*/
  if(onlyDirs&&(fi.m_type!=CFileInfo::DIR))
    return("");
  string str;

/*
** in line sind die grafischen Elemente von Funktionen
** aus höheren Rekursionsstufen gespeichert, um die Äste
** korrekt zeichnen zu können.
*/
  str+=line;

/*
** Wenn letzter Eintrag, dann Astende zeichnen,
** andernfalls eine Verzweigung
*/
  if(last)
    str+="<img src=\"t_branchend.gif\" align=\"middle\" width=\""+
         toString(imgsize)+"\" />";
```

Listing 486: Überladene Variante von fileInfoToHtml für rekursive Aufrufe

```
  else
    str+="<img src=\"t_branch.gif\" align=\"middle\" width=\""+
         toString(imgsize)+"\" />";

/*
** Je nachdem, ob Datei oder Verzeichnis, die
** Grafik t_file.gif oder t_dir.gif zeichnen
*/
  if(fi.m_type==CFileInfo::DIR)
    str+="<img src=\"t_dir.gif\" align=\"middle\" width=\""+
         toString(imgsize)+"\" />";
  else
    str+="<img src=\"t_file.gif\" align=\"middle\" width=\""+
         toString(imgsize)+"\" />";

/*
** Eintragsnamen und -größe schreiben
*/
  str+=fi.m_name+" ("+
       CHugeNumber(fi.m_size.m_lowsize,
                   fi.m_size.m_highsize).getAsPointedNumber()+")";
  str+="<br />\n";

/*
** Für tiefere Rekursionsstufe line um die entsprechende
** Astgrafik ergänzen
*/
  if(last)
    line+="<img src=\"t_space.gif\" align=\"middle\" width=\""+
          toString(imgsize)+"\" />";
  else
    line+="<img src=\"t_line.gif\" align=\"middle\" width=\""+
          toString(imgsize)+"\" />";

/*
** Besitzt Eintrag Dateien oder Unterverzeichnisse?
** => fileInfoToHtml für jeden Eintrag aufrufen
*/
  if(fi.m_filelist&&(fi.m_filelist->size()>0))
    for(unsigned int i=0; i<fi.m_filelist->size(); i++)
      str+=fileInfoToHtml((*fi.m_filelist)[i],
                          line,
```

Listing 486: Überladene Variante von fileInfoToHtml für rekursive Aufrufe (Forts.)

```
                        onlyDirs,
                        imgsize,
                        (i==(fi.m_filelist->size()-1)));

  return(str);
}
```

Listing 486: Überladene Variante von fileInfoToHtml für rekursive Aufrufe (Forts.)

Die Funktionen finden Sie auf der CD in den `FileFunctions`-Dateien.

Zur Darstellung der Dateigröße wird die Klasse `CHugeNumber` aus Rezept 119 verwendet.

Damit die verwendeten Grafiken korrekt dargestellt werden, müssen sich im selben Verzeichnis mit dem erzeugten HTML-Dokument die Grafiken `t_dir.gif`, `f_file.gif`, `t_branch.gif`, `t_branchend.gif`, `t_space.gif` und `t_line.gif` aus dem `images`-Ordner auf der CD befinden.

97 Wie kann ein Dateifilter eingesetzt werden?

Unser Dateifilter soll in der Lage sein, Dateien anhand bestimmter Kriterien auszusortieren. Dazu sollen Attribute wie Dateiname, Dateigröße, Erstellungsdatum berücksichtigt werden.

CFileFilter

Wir implementieren dazu eine Klasse `CFileFilter` mit folgenden Attributen:

- `m_namePattern` – auf den Dateinamen anzuwendendes Muster (Standard: `?*`)
- `m_caseSensitivity` – boolescher Wert, der darüber entscheidet, ob Groß- und Kleinschreibung berücksichtigt werden soll (`true`) oder nicht (`false`) (Standard: `false`)
- `m_types` – welche Dateitypen sollen berücksichtigt werden; Verzeichnisse (`DIR`), Dateien (`FILE`) oder beide (`BOTH`.) (Standard: `BOTH`)
- `m_minCreated` – frühestes Erstellungsdatum (Standard: 01.01.0001 00:00:00:000)
- `m_maxCreated` – spätestes Erstellungsdatum (Standard: 31.12.3000 23:59:59:999)
- `m_minModified` – frühestes Datum einer Veränderung (Standard: 01.01.0001 00:00:00:000)
- `m_maxModified` – spätestes Datum einer Veränderung (Standard: 31.12.3000 23:59:59:999)

```
class CFileFilter {
public:
  enum FTYPE {DIR=1, FILE=2, BOTH=3};
private:
  bool m_caseSensitivity;
  CMoment m_minCreated;
  CMoment m_maxCreated;
  CMoment m_minModified;
  CMoment m_maxModified;
  CFileSize m_minSize;
  CFileSize m_maxSize;
  std::string m_namePattern;
  FTYPE m_types;
  CAutomat m_nameAutomat;
};
```

Listing 487: Die Klassendefinition von CFileFilter

Um das Suchmuster einsetzen zu können wird der Automat aus Rezept 18 verwendet. Die Daten werden mit der Klasse `CMoment` aus Rezept 61 verwaltet.

Konstruktor

Der Konstruktor, der die Attribute mit den entsprechenden Standard-Werten versieht, ist im Folgenden aufgeführt.

```
  CFileFilter(std::string pattern="?*",
              bool casesensi=false,
              FTYPE types=BOTH,
              CFileSize minsize=CFileSize(0,0),
              CFileSize maxsize=CFileSize(0xffffffff,0xffffffff),
              CMoment minmodi=CMoment(1,1,1,0,0,0,0),
              CMoment maxmodi=CMoment(3000,12,31,23,59,59,999),
              CMoment mincrea=CMoment(1,1,1,0,0,0,0),
              CMoment maxcrea=CMoment(3000,12,31,23,59,59,999))

  : m_namePattern(pattern),
    m_caseSensitivity(casesensi),
    m_types(types),
    m_minSize(minsize),
    m_maxSize(maxsize),
```

Listing 488: Der Konstruktor der Klasse CFileFilter

```
    m_minModified(minmodi),
    m_maxModified(maxmodi),
    m_minCreated(mincrea),
    m_maxCreated(maxcrea),
    m_nameAutomat((casesensi)?pattern:toLowerString(pattern))
  {}
```

Listing 488: Der Konstruktor der Klasse CFileFilter (Forts.)

Zugriffsmethoden

Falls einzelne Attribute nach Erstellen eines Objekts noch verändert werden müssen, existieren entsprechende Zugriffsmethoden.

```
  void setPattern(const std::string &p) {
    m_namePattern=p;
  }

  void setCaseSensitivity(bool cs) {
    m_caseSensitivity=cs;
  }

  void setTypes(FTYPE f) {
    m_types=f;
  }

  void setMinSize(const CFileSize &fs) {
    m_minSize=fs;
  }

  void setMaxSize(const CFileSize &fs) {
    m_maxSize=fs;
  }

  void setMinCreated(const CMoment &m) {
    m_minCreated=m;
  }

  void setMaxCreated(const CMoment &m) {
    m_maxCreated=m;
  }
```

Listing 489: Die Zugriffsmethoden von CFileFilter

```
void setMinModified(const CMoment &m) {
  m_minModified=m;
}

void setMaxModified(const CMoment &m) {
  m_maxModified=m;
}

std::string getPattern() const {
  return(m_namePattern);
}

CFileSize getMinSize() const {
  return(m_minSize);
}

CFileSize getMaxSize() const {
  return(m_maxSize);
}

CMoment getMinCreated() const {
  return(m_minCreated);
}

CMoment getMaxCreated() const {
  return(m_maxCreated);
}

CMoment getMinModified() const {
  return(m_minModified);
}

CMoment getMaxModified() const {
  return(m_maxModified);
}

bool getCaseSensitivity() const {
  return(m_caseSensitivity);
}
```

Listing 489: Die Zugriffsmethoden von CFileFilter (Forts.)

```
  FTYPE getTypes() const {
    return(m_types);
  }
```

Listing 489: Die Zugriffsmethoden von CFileFilter (Forts.)

match

Um zu überprüfen, ob eine Datei die Kriterien erfüllt, existieren verschiedene Varianten der `match`-Funktion.

Die einfachste Variante überprüft lediglich den Dateinamen:

```
bool CFileFilter::match(const string &n) const {

/*
** Groß- und Kleinschreibung nicht beachten?
** => Entsprechende CAutomat-Methode aufrufen
*/
  if(m_caseSensitivity)
    return(m_nameAutomat.match(n));
  else
    return(m_nameAutomat.matchCI(n));
}
```

Listing 490: Die match-Funktion für Dateinamen

Die aufwändigere Variante von `match` erwartet ein `CFileInfo`-Objekt als Funktionsparameter. Sie wertet alle Kriterien aus und liefert als booleschen Wert zurück, ob das `CFileInfo`-Objekt den Anforderungen genügt.

```
bool CFileFilter::match(const CFileInfo &fi) const{

/*
** Prüfen, ob gewünschter Dateityp
*/
  switch(fi.m_type) {
    case CFileInfo::DIR:
      if(!(m_types&DIR))
        return(false);
```

Listing 491: match-Funktion für CFileInfo-Objekte

```
      break;

    case CFileInfo::FILE:
      if(!(m_types&FILE))
        return(false);
      break;
  }

/*
** Dateinamen überprüfen
** Groß- und Kleinschreibung nicht beachten?
** => Entsprechende CAutomat-Methode aufrufen
*/
  bool matches;
  if(m_caseSensitivity)
    matches=m_nameAutomat.match(fi.m_name);
  else
    matches=m_nameAutomat.matchCI(fi.m_name);
  if(!matches)
    return(false);

/*
** Dateigröße überprüfen
*/
  if((fi.m_size<m_minSize)||(fi.m_size>m_maxSize))
    return(false);

/*
** Erstellungsdatum überprüfen
*/
  if((fi.m_created<m_minCreated)||(fi.m_created>m_maxCreated))
    return(false);

/*
** Datum der letzten Änderung überprüfen
*/
  if((fi.m_modified<m_minModified)||(fi.m_modified>m_maxModified))
    return(false);

  return(true);
}
```

Listing 491: match-Funktion für CFileInfo-Objekte (Forts.)

Stream-Operatoren

Damit die Klasse auch mit den binären Strömen aus Rezept 106 zusammenarbeitet, implementieren wir noch die Operatoren << und >> für `CBinaryOStream` und `CBinaryIStream`:

Bei `CFileFilter` wird nicht der gesamte Automat gespeichert, sondern nur das Muster, aus dem der Automat wieder generiert werden kann. Deswegen muss beim Laden auch die `rebuild`-Methode des Automaten aufgerufen werden.

Damit für den benutzerdefinierten Typ `FTYPE` nicht auch Stream-Operatoren überladen werden müssen, wird das Attribut `m_types` direkt über die Methoden `write` und `read` gespeichert und geladen.

Die Stream-Operatoren müssen als Freunde der Klasse deklariert werden.

```
CBinaryOStream &operator<<(CBinaryOStream &os, const CFileFilter &ff) {
  os << ff.m_caseSensitivity;
  os << ff.m_minCreated;
  os << ff.m_maxCreated;
  os << ff.m_minModified;
  os << ff.m_maxModified;
  os << ff.m_minSize;
  os << ff.m_maxSize;
  os << ff.m_namePattern;
  os.write(&ff.m_types, sizeof(ff.m_types));
  return(os);
}

//*********

CBinaryIStream &operator>>(CBinaryIStream &is, CFileFilter &ff) {
  is >> ff.m_caseSensitivity;
  is >> ff.m_minCreated;
  is >> ff.m_maxCreated;
  is >> ff.m_minModified;
  is >> ff.m_maxModified;
  is >> ff.m_minSize;
  is >> ff.m_maxSize;
  is >> ff.m_namePattern;
  is.read(&ff.m_types, sizeof(ff.m_types));
```

Listing 492: Die Stream-Operatoren für CFileFilter

```
  ff.m_nameAutomat.rebuild(ff.m_namePattern);
  return(is);
}
```

Listing 492: Die Stream-Operatoren für CFileFilter (Forts.)

Die Klasse `CFileFilter` finden Sie auf der CD in den `CFileFilter`-Dateien.

98 Wie kann mit Mustern in Windows-Syntax gearbeitet werden?

Obwohl die Syntax unseres Automaten aus Rezept 18 viel mächtiger ist als die Joker-Funktion in der Windows-Suche, kann es in einigen Bereichen durchaus Sinn machen, die einfachere Windows-Syntax zu benutzen.

convertWinPattern

Die Klasse `CFileFilter` wird durch die statische Methode `convertWinPattern` ergänzt. Sie bekommt das in Windows-Syntax formulierte Muster übergeben und liefert ein für unseren Automaten verständliches Muster zurück.

```
string CFileFilter::convertWinPattern(const string &p) {
  string str;

/*
** Alle Zeichen des Musters durchlaufen
*/
  for(size_t i=0; i<p.size(); i++) {

/*
** Oder-Verknüpfung umwandeln
*/
    if((p[i]==';')&&(i!=p.size()-1))
      str+="|";

/*
** Hüllenbildung umwandeln
*/
    else if(p[i]=='*')
      str+="?*";
```

Listing 493: Die statische Methode convertWinPattern

```
/*
** Positive Hülle kodieren
*/
    else if(p[i]=='+')
      str+="~+";

/*
** Sonderzeichen kodieren
*/
    else if(p[i]=='~')
      str+="~~";

/*
** " und abschließendes ; ignorieren
*/
    else if((p[i]=='\"')||
            (p[i]==';'));

/*
** Eckige Klammern kodieren
*/
    else if((p[i]=='[')||
            (p[i]==']')) {
      str+="~";
      str+=p[i];
    }

/*
** Geschweifte Klammern kodieren
*/
    else if((p[i]=='{')||
           (p[i]=='}')) {
      str+="~";
      str+=p[i];
    }

/*
** Zeichen unverändert übernehmen
*/
    else
```

Listing 493: Die statische Methode convertWinPattern (Forts.)

```
      str+=p[i];
  }
  return(str);
}
```

Listing 493: Die statische Methode convertWinPattern (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

99 Wie kann nach Dateien gesucht werden?

Die Suche nach Dateien kann durch den Dateifilter aus Rezept 97 stark vereinfacht werden.

findFiles

Es wird dazu eine Funktion `findFiles` geschrieben, die folgende Parameter besitzt:

- `name` – Name des Verzeichnisses, in dem gesucht werden soll
- `filefilter` – das einzusetzende `FileFilter`-Objekt
- `filelist` – die Datei-Liste, in der die gefundenen Dateien abgelegt werden

Die Funktion macht von `getDir` aus Rezept 85 Gebrauch.

```
void findFiles(const string &name,
               const CFileFilter &filefilter,
               CFileInfo::filelist_type &filelist) {

  CFileInfo::filelist_type direntries;

/*
** Verzeichnis-Einträge auslesen
*/
  getDir(name,direntries);

/*
** Alle Einträge bearbeiten
*/
  for(CFileInfo::filelist_type::iterator iter=direntries.begin();
      iter!=direntries.end();
```

Listing 494: Die Funktion findFiles

```
      ++iter) {

/*
** Eintrag mit dem FileFilter prüfen und
** ggfs. zu den Suchergebnissen hinzufügen
*/
    if(filefilter.match(*iter))
      filelist.push_back(*iter);

/*
** Ist Eintrag ein Verzeichnis?
** => findFiles rekursiv aufrufen
*/
    if(iter->m_type==CFileInfo::DIR)
      findFiles(name+"/"+iter->m_name,filefilter,filelist);
  }
}
```

Listing 494: Die Funktion findFiles (Forts.)

Vielleicht wurde die Verzeichnisstruktur bereits mit `getCompleteDir` aus Rezept 93 eingelesen. Ein weiteres Durchforsten der Verzeichnisse auf dem Datenträger wäre damit unnötige Zeitverschwendung.

Wir implementieren deswegen eine weitere Variante von `findFiles`, der anstelle eines Verzeichnisnamens ein `CFileInfo`-Objekt übergeben wird, welches den Kopf einer Verzeichnisstruktur bildet:

```
void findFiles(const CFileInfo &structure,
               const CFileFilter &filefilter,
               CFileInfo::filelist_type &filelist) {

/*
** Genügt aktuelle FileInfo den Kriterien?
*/
  if(filefilter.match(structure))
    filelist.push_back(structure.getShallowCopy());

/*
** Alle Einträge bearbeiten
*/
```

Listing 495: Die Methode findFiles für eine eingelesene Verzeichnisstruktur

```
  if(structure.m_filelist!=0) {
    for(CFileInfo::filelist_type::iterator iter=
                                  structure.m_filelist->begin();
        iter!=structure.m_filelist->end();
        ++iter) {

/*
** Ist Eintrag ein Verzeichnis?
** => findFiles rekursiv aufrufen
*/
      if(iter->m_type==CFileInfo::DIR)
        findFiles(*iter,filefilter,filelist);

/*
** Andernfalls Eintrag mit dem FileFilter prüfen und
** ggfs. zu den Suchergebnissen hinzufügen
*/
      else
        if(filefilter.match(*iter))
          filelist.push_back((*iter).getShallowCopy());
    }
  }
}
```

Listing 495: Die Methode findFiles für eine eingelesene Verzeichnisstruktur (Forts.)

Die Funktionen finden Sie auf der CD in den `FileFunctions`-Dateien.

100 Wie kann ein Suchergebnis dargestellt werden?

Die Darstellung wird wieder mit Hilfe von HTML umgesetzt. Erzeugt werden soll eine Formatierung wie in Abbildung 40 dargestellt.

E:/CB/Struktur/?*.cpp

FileFunctions.cpp	9.120	E:/CB/Struktur/a1/b1/c1	10.06.2003 11:13:56
FileSpecifics.cpp	3.558	E:/CB/Struktur/a2	09.06.2003 10:04:11
HTMLFunctions.cpp	505	E:/CB/Struktur	06.06.2003 13:22:09

3 Objekte, 13.183 Bytes

Abbildung 40: Darstellung eines Suchergebnisses

searchResultsToHtml

Die Funktion besitzt dieselben Parameter wie `findFiles` aus Rezept 99, wandelt sie doch deren Ergebnisse in HTML um.

```
string searchResultsToHtml(const std::string &name,
                           const CFileFilter &ff,
                           const CFileInfo::filelist_type &fl) {

  string str;

/*
** Pfad als Überschrift mit Linie schreiben
*/
  str+="<h2>"+name+"/"+ff.getPattern()+"<hr /></h2>\n";

  CFileSize fs;

/*
** Tabelle erzeugen und alle Einträge durchlaufen
*/
  str+="<table>\n";
  for(CFileInfo::filelist_type::const_iterator iter=fl.begin();
      iter!=fl.end();
      ++iter) {
    str+="<tr><td>";

/*
** Entsprechend des Eintrag-Typs die Grafik
** t_dir.gif oder t_file.gif verwenden
*/
    switch(iter->m_type) {
      case CFileInfo::DIR:
        str+="<img src=\"t_dir.gif\" width=\"20\" />";
        break;

      case CFileInfo::FILE:
        str+="<img src=\"t_file.gif\" width=\"20\" />";
        break;
    }

/*
```

Listing 496: Die Funktion searchResultsToHtml

```
** Name des Eintrags schreiben
*/
    str+="</td><td>";
    str+=iter->m_name;
    str+="</td><td width=\"10\">";

/*
** Eintrag vom Typ FILE?
** => Dateigröße schreiben und aufaddieren
*/
    if(iter->m_type==CFileInfo::FILE) {
      str+="</td><td>";
      str+=CHugeNumber(iter->m_size.m_lowsize,
                  iter->m_size.m_highsize).getAsPointedNumber();
      fs+=iter->m_size;
    }
    else
      str+="</td><td>";

/*
** Pfad schreiben
*/
    str+="</td><td width=\"10\"></td><td>";
    str+=iter->m_path;

/*
** Datum der letzten Änderung schreiben
*/
    str+="</td><td width=\"10\"></td><td>";
    str+=iter->m_modified.getDDMMYYYYHHMMSS();

/*
** Tabellenzelle und -zeile beenden
*/
    str+="</td></tr>\n";
  }

/*
** Tabelle beenden und String zurückliefern
*/
  str+="</table>\n";
```

Listing 496: Die Funktion searchResultsToHtml (Forts.)

```
/*
** Unter einem Strich die Anzahl der gefundenen Objekte
** sowie deren gesamte Speichergröße schreiben
*/
  str+="<hr />"+toString(static_cast<unsigned long>(fl.size()))+"
       Objekte, ";
  str+=CHugeNumber(fs.m_lowsize,
                   fs.m_highsize).getAsPointedNumber();
  str+=" Bytes";
  return(str);
}
```

Listing 496: Die Funktion searchResultsToHtml (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

Damit die verwendeten Grafiken korrekt dargestellt werden, müssen sich im selben Verzeichnis mit dem erzeugten HTML-Dokument die Grafiken `t_dir.gif` und `f_file.gif` aus dem `images`-Ordner auf der CD befinden.

101 Wie können binäre Dateien einfacher gehandhabt werden?

Um mit den binären Streams von C++ etwas einfacher arbeiten zu können, wollen wir eine Klasse `CFile` schreiben, die das lästige Umwandeln des Typs bei den Methoden `read` und `write` eliminiert.

CFile

Die Klasse besitzt folgende Attribute:

- `m_file` – Zeiger auf den Datenstrom
- `m_filename` – der Name der geöffneten Datei
- `m_size` – die Größe der geöffneten Datei

Eine eingebettete Fehler-Klasse `file_error` ist ebenfalls vorhanden.

```
class CFile {
  private:
    std::fstream *m_file;
    std::string m_filename;
    std::streampos m_size;

  public:
    class file_error {
      public:
        const std::string name;
        file_error(const std::string &n) :name(n) {};
    };
};
```

Listing 497: Die Klassendefinition von CFile

Zugriffsmethoden

Bei der Zugriffsmethode `size` wurde auf die bisher übliche Vorsilbe »get« verzichtet, um sich den Containern der STL anzunähern.

```
unsigned long size() {
  return(m_size);
}
std::string getFileName(void) {
  return(m_filename);
}
```

Listing 498: Die Zugriffsmethoden von CFile

Konstruktor

Der dreiparametrige Konstruktor kann wegen der Standard-Werte der beiden letzten Parameter auch als Standard-Konstruktor eingesetzt werden. Deswegen wird er als `explicit` deklariert.

Die Bedeutung der Parameter ist identisch mit denen von `open`.

```
CFile::CFile()
: m_file(0)
```

Listing 499: Die Konstruktoren von CFile

```
{}

//*********

CFile::CFile(const string &name, bool create, bool erase)
: m_file(0) {
  if(!open(name,create))
    throw file_error(m_filename);
}
```

Listing 499: Die Konstruktoren von CFile (Forts.)

Destruktor

```
CFile::~CFile() {
  if(m_file)
    delete(m_file);
}
```

Listing 500: Der Destruktor von CFile

open

Die open-Methode ermittelt die Dateigröße, indem sie den Dateipositionszeiger an das Dateiende setzt und ihn dann ausliest.

Die Dateigröße hätte auch mit der Funktion `getFileInfo` aus Rezept 84 bestimmt werden können, aber das hätte ein Zugreifen auf plattformabhängige Funktionen bedeutet.

Das Argument `create` entscheidet, ob eine nicht existierende Datei angelegt werden soll, und besitzt den Standardwert `true`.

Über `erase` kann entschieden werden, ob eine bereits bestehende Datei gelöscht werden soll. Standard ist `false`.

```
bool CFile::open(const string &name, bool create, bool erase) {

/*
** Existierende Datei öffnen bzw. löschen
```

Listing 501: Die Methode open von CFile

```
*/
  if(erase)
    m_file=new fstream(name.c_str(),
                       ios::in|ios::out|ios::binary|ios::trunc);
  else
    m_file=new fstream(name.c_str(), ios::in|ios::out|ios::binary);

/*
** Soll bei nicht existenter Datei eine neue angelegt werden?
** => Entsprechende Überprüfungen und Schritte einleiten
*/
  if(create) {

/*
** Fehlgeschlagen? Vielleicht Datei nicht existent
** ==> Neue Datei öffnen
*/
    if(!m_file->is_open()) {
      delete(m_file);
      m_file=new fstream(name.c_str(),
                         ios::in|ios::out|ios::binary|ios::trunc);
    }
  }

/*
** Wieder fehlgeschlagen?
** => Abbruch
*/
  if(!m_file->is_open())
    return(false);

  m_filename=name;

/*
** Dateipositionszeiger ans Dateiende setzen und
** Dateigröße ermitteln
*/
  m_file->seekg(0,ios::end);
  if(m_file->fail())
    return(false);
```

Listing 501: Die Methode open von CFile (Forts.)

```
  m_size=m_file->tellg();
  if(m_file->fail())
    return(false);

/*
** Dateipositionszeiger an Dateianfang setzen
*/
  m_file->seekg(0,ios::beg);
  if(m_file->fail())
    return(false);

  return(true);
}
```

Listing 501: Die Methode open von CFile (Forts.)

readBlock

Es existieren zwei `readBlock`-Methoden, die erste liest ab der aktuellen Position des Dateipositionszeigers, die zweite bietet die Möglichkeit, die Position frei zu wählen. Sie benutzen `void`-Zeiger und ersparen es dem Benutzer dadurch, den eigenen Zeigertypen umzuwandeln.

```
void CFile::readBlock(void *addr, streampos size) {

/*
** Block lesen
*/
  m_file->read(reinterpret_cast<char*>(addr), size);
  if(m_file->fail())
    throw file_error(m_filename);
}

void CFile::readBlock(void *addr,
                      streampos size,
                      streampos pos) {

/*
** Dateipositionszeiger setzen
*/
  m_file->seekg(pos,ios::beg);
```

Listing 502: Die readBlock-Methoden von CFile

```
  if(m_file->fail())
    throw file_error(m_filename);

/*
** Block lesen
*/
  m_file->read(reinterpret_cast<char*>(addr),size);
  if(m_file->fail())
    throw file_error(m_filename);
}
```

Listing 502: Die readBlock-Methoden von CFile (Forts.)

writeBlock

Die `writeBlock`-Methoden sind die Gegenstücke zu `readBlock` und arbeiten ebenfalls mit `void`-Zeigern.

```
void CFile::writeBlock(const void *addr, streampos size) {

/*
** Block schreiben
*/
  m_file->write(reinterpret_cast<const char*>(addr), size);
  if(m_file->fail())
    throw file_error(m_filename);
}

void CFile::writeBlock(const void *addr,
                       streampos size,
                       streampos pos) {

/*
** Dateipositionszeiger setzen
*/
  m_file->seekp(pos,ios::beg);
  if(m_file->fail())
    throw file_error(m_filename);

/*
** Block schreiben
*/
```

Listing 503: Die writeBlock-Methoden von CFile

```
  m_file->write(reinterpret_cast<const char*>(addr), size);
  if(m_file->fail())
    throw file_error(m_filename);
}
```

Listing 503: Die writeBlock-Methoden von CFile (Forts.)

expand

Die `expand`-Methode bietet die Möglichkeit, die Dateigröße zu vergrößern. Leider gibt es bei den Streams keine Möglichkeit, dies explizit zu tun. Es bleibt nichts anderes übrig, als an das Ende der Datei so viele Daten zu schreiben, bis die gewünschte Größe erreicht ist.

```
void CFile::expand(std::streampos size) {

/*
** Dateipositionszeiger ans Dateiende setzen
*/
  m_file->seekg(0,ios::end);
  if(m_file->fail())
    throw file_error(m_filename);

/*
** So viele Zeichen schreiben, wie zur gewünschten
** Vergrößerung notwendig sind
*/
  char c='\0';
  for(streampos count=0; count<(size-m_size); count+=1)
    writeBlock(&c,1);
  if(m_file->fail())
    throw file_error(m_filename);

/*
** Neue Dateigröße ermitteln
*/
  m_size=m_file->tellg();
}
```

Listing 504: Die expand-Methode von CFile

Die Klasse `CFile` finden Sie auf der CD in den `CFile`-Dateien.

102 Wie kann eine Datei mit Index-Operator angesprochen werden?

Um den Inhalt einer Datei mit Index-Operator ansprechen zu können, implementieren wir die Klasse `CFileArray`. Sie benutzt die Klassen `CFile` aus Rezept 101 und `CMemory` aus Rezept 138.

Die Schwierigkeit liegt in der Natur des Index-Operators. Man kann nicht feststellen, ob der Index-Operator dazu verwendet wurde, um Daten zu lesen oder Daten zu beschreiben. In unserem Fall ist diese Information jedoch sehr wichtig, denn wir müssen eine eventuelle Änderung der Daten in die Datei zurückschreiben. Wir verwenden dazu eine Proxy-Klasse, die als Schnittstelle zwischen den Daten der `CFileArray`-Klasse und dem Benutzer fungiert.

Ein Objekt von `CProxy` repräsentiert genau ein `char`-Element der Datei.

CProxy

Die `CProxy`-Klasse besitzt nur zwei Attribute: `m_index` ist der Index des repräsentierten `char`-Elements in der Datei und `m_fileArray` ist ein Verweis auf das die Datei verwaltende `CFileArray`-Objekt.

```
class CProxy {
  friend class CFileArray;
  private:
    std::streampos m_index;
    CFileArray &m_fileArray;
};
```

Listing 505: Die Klassendefinition von CProxy

Konstruktor

Der Konstruktor ist denkbar einfach aufgebaut. Er ist als privat deklariert, damit nur die befreundete `CFileArray`-Klasse ihn einsetzen kann.

```
CProxy(CFileArray &d, std::streampos i)
  : m_fileArray(d), m_index(i)
{}
```

Listing 506: Der Konstruktor von CProxy

operator char

`operator char` sorgt dafür, dass ein `CProxy`-Objekt auch als `char` benutzt werden kann. Es holt aus dem dazugehörigen `CFileArray`-Objekt das Zeichen mit dem entsprechenden Index.

Dies geschieht über die Methode `CharRef`, die eines der Herzstücke der `CFileArray`-Klasse ist.

```
operator char() const {
  return(m_fileArray.CharRef(m_index));
}
```

Listing 507: Die operator char-Methode von CProxy

Zuweisungsoperatoren

Die Zuweisungsoperatoren beschreiben das repräsentierte `char`-Element über `CFileArray.CharRef` mit dem entsprechenden Wert.

Wichtig ist, dass hier `m_changed` auf true gesetzt wird, damit die geänderten Daten später in die Datei übertragen werden.

```
CProxy &operator=(const CProxy &p){
  m_fileArray.CharRef(m_index)=p.m_fileArray.CharRef(p.m_index);
  m_fileArray.m_changed=true;
  return(*this);
}

//*********

CProxy &operator=(char c){
  m_fileArray.CharRef(m_index)=c;
  m_fileArray.m_changed=true;
  return(*this);
}
```

Listing 508: Die Zuweisungsoperatoren von CProxy

Ausgabeoperator

Damit ein von `CProxy` repräsentiertes Zeichen auch problemlos ausgegeben werden kann, wird die `operator`<<-Funktion überladen.

```
friend std::ostream &operator<<(std::ostream &ostr, const CProxy &p) {
  ostr << (static_cast<char>(p));
  return(ostr);
}
```

Listing 509: Der Ausgabeoperator von CProxy

Die Klasse `CFile` finden Sie eingebettet in die Klasse `CFileArray` auf der CD in den `CFileArray`-Dateien.

CFileArray

Kommen wir nun zur `CFileArray`-Klasse. Sie besitzt folgende Attribute:

- `m_mem` – Speicher, in dem ein Teil der Datei abgebildet wird, als `CMemory`-Objekt
- `m_file` – die Datei als `CFile`-Objekt
- `m_pufferSize` – die Größe des für das Dateiabbild verwendeten Speichers
- `m_size` – die Größe der Datei
- `m_blockPos` – Position des im Speicher befindlichen Blocks in der Datei
- `m_blockSize` – Größe des Blocks im Speicher
- `m_changed` – boolescher Wert, der nachhält, ob Änderungen am Speicherblock (und damit an der Datei) vorgenommen wurden
- `npos` – Wert, der für eine ungültige Position steht (ist für Suchfunktionen wichtig)

```
class CFileArray
{
  private:
    CMemory<char> m_mem;
    CFile m_file;
    std::streampos m_pufferSize;
    bool m_changed;
    std::streampos m_size;
    std::streampos m_blockPos;
    std::streampos m_blockSize;

  public:
```

Listing 510: Die Klassendefinition von CFileArray

```
    static std::streampos npos;

    class CProxy;
    friend class CFileArray::CProxy;

    class file_error {
      public:
        const std::string name;
        file_error(const std::string &n) :name(n) {}
    };
};
```

Listing 510: Die Klassendefinition von CFileArray (Forts.)

Initialisierung des statischen Attributs

Das statische Attribut `npos` wird auf den nicht auftretenden Wert -1 gesetzt:

```
streampos CFileArray::npos=-1;
```

Listing 511: Initialisierung von npos

Zugriffsmethode

`CFileArray` besitzt nur eine Zugriffsmethode, bei der aus Gründen der Ähnlichkeit zur STL wieder die Vorsilbe »get« fehlt:

```
std::streampos size(void) const{
  return(m_size);
}
```

Listing 512: Die Methode size von CFileArray

Konstruktor

Der Konstruktor initialisiert die Attribute und lädt den Anfang der Datei als Ausschnitt in den Speicher. Er besitzt folgende Parameter:

- `name` – Name der zu öffnenden Datei
- `create` – ein boolescher Wert, der entscheidet, ob eine nicht existierende Datei neu angelegt werden soll (true) oder nicht (false) (Standard: true)
- `puffsize` – Größe des Dateipuffers im Speicher (Standard: 50000)

```
CFileArray::CFileArray(const string &name,
                       bool create,
                       unsigned long puffsize)
: m_mem(puffsize), m_file(name,create), m_pufferSize(puffsize) {
  m_changed=false;
  m_blockPos=0;
  m_size=m_file.size();

/*
** Dateianfang in Speicher lesen
*/
  readBlock();
}
```

Listing 513: Der Konstruktor von CFileArray

Destruktor

Der Destruktor sorgt dafür, dass alle geänderten Daten vor der Objekt-Zerstörung gesichert werden.

```
CFileArray::~CFileArray() {

/*
** Noch geänderte Daten im Pufferspeicher?
** => Daten vor Zerstörung in Datei schreiben
*/
  if(m_changed) {
    writeBlock();
    m_changed=false;
  }
}
```

Listing 514: Der Destruktor von CFileArray

Index-Operator

Der Index-Operator liefert die Daten in Form von `CProxy`-Objekten.

```
CFileArray::CProxy CFileArray::operator[](unsigned long b) {
```

Listing 515: Der Index-Operator von CFileArray

```
/*
** Ein CProxy-Objekt für das gewünschte char-Element zurückliefern
*/
  return(CProxy(*this,b));
}
```

Listing 515: Der Index-Operator von CFileArray (Forts.)

readBlock

`readBlock` liest an einer bestimmten Stelle in der Datei einen entsprechend großen Bereich in den Pufferspeicher.

```
void CFileArray::readBlock() {

/*
** Liegen in der Datei ab Position m_blockPos weniger
** Daten, als der Puffer groß ist?
*/
  if((m_blockPos+m_pufferSize)>m_size)

/*
** JA: Blockgröße auf Größe der restlichen Daten
** in der Datei setzen
*/
    m_blockSize=m_size-m_blockPos;
  else

/*
** NEIN: Blockgröße auf Größe des Pufferspeichers setzen
*/
    m_blockSize=m_pufferSize;

/*
** Daten in Pufferspeicher lesen
*/
  m_file.readBlock(m_mem,m_blockSize,m_blockPos);
}
```

Listing 516: Die Methode readBlock von CFileArray

writeBlock

Die `writeBlock`-Methode speichert im Pufferspeicher geänderte Daten zurück in die Datei.

```
void CFileArray::writeBlock() {

/*
** Pufferspeicher an die entsprechende Position
** in der Datei schreiben
*/
  m_file.writeBlock(m_mem,m_blockSize,m_blockPos);
}
```

Listing 517: Die Methode writeBlock von CFileArray

CharRef

Die `CharRef`-Methode ist der Kern von `CFileArray`. Über sie wird ein char-Element der Datei angesprochen.

Sollte das gewünschte Element nicht in dem Bereich liegen, der sich aktuell im Speicher befindet, dann sorgt `CharRef` dafür, dass der aktuelle Bereich gegebenenfalls abgespeichert und ein neuer Bereich mit dem gewünschten Element geladen wird.

```
char &CFileArray::CharRef(streampos b) {

/*
** Wird auf Daten hinter dem Dateiende zugegriffen?
** => Datei bis zu dieser Stelle vergrößern
*/
  if(b>=m_size) {
    m_file.expand(b+1L);
    m_size=m_file.size();
  }

/*
** Liegt gewünschtes char-Element in dem Datei-Bereich,
** der sich im Pufferspeicher befindet?
** => char-Element zurückliefern
*/
```

Listing 518: Die CharRef-Methode von CFileArray

```
  if((b>=m_blockPos)&&(b<(m_blockPos+m_blockSize)))
    return(m_mem[b-m_blockPos]);
  else {

/*
** Sollte gewünschtes char-Element nicht im Pufferspeicher
** liegen, dann:
** Befinden sich im Pufferspeicher geänderte Daten?
** => Abspeichern
*/
    if(m_changed) {
      writeBlock();
      m_changed=false;
    }

/*
** Liegt gewünschtes char-Element dicht am Dateianfang?
** => Neuer Block liegt am Dateianfang
*/
    if((m_pufferSize/2)>=b)
      m_blockPos=0;
    else

/*
** Wenn nicht, dann Blockposition so wählen, dass
** sich das gewünschte char-Element in der Mitte des
** Blocks befindet
*/
      m_blockPos=b-(m_pufferSize/2);

/*
** Block einlesen und gewünschtes char-Element zurückliefern.
*/
    readBlock();
    return(m_mem[b-m_blockPos]);
  }
}
```

Listing 518: Die CharRef-Methode von CFileArray (Forts.)

Die Klasse `CFileArray` finden Sie auf der CD in den `CFileArray`-Dateien.

103 Wie kann Text in einer Datei gesucht werden?

Um effizient in einer Datei suchen zu können, sollte auf sie ein wahlfreier Zugriff möglich sein. Über die binären Dateien ist ein wahlfreier Zugriff grundsätzlich möglich, aber für jedes Zeichen den Dateipositionszeiger neu zu setzen kann je nach Suchmethode viel Zeit kosten.

Um diesem Problem etwas zu entgehen, benutzen die nachstehenden Suchfunktionen für den Dateizugriff die Klasse `CFileArray` aus Rezept 102. Sie puffert die Datei (oder je nach Dateigröße Teile davon) im Speicher und reduziert die langsamen Zugriffe auf den Festspeicher.

findFirstInFile

Die Funktion `findFirstInFile` kommt in zwei Versionen, die eine erwartet ein bereits bestehendes `CFileArray`-Objekt, die andere erstellt es selbst.

```
streampos findFirstInFile(const string &name,
                          const string &search,
                          streampos pos){
  try{

/*
** Datei öffnen und überladene Funktion aufrufen
*/
    CFileArray file(name,false);
    return(findFirstInFile(file,search,pos));
  }
  catch(...) {
    return(CFileArray::npos);
  }
}

//*********

streampos findFirstInFile(CFileArray &file,
                          const string &search,
                          streampos pos){

/*
** Größe der Datei und des
** Suchstrings ermitteln
```

Listing 519: Die Funktion findFirstInFile

```
*/
  streampos fsize=file.size();
  streampos ssize=search.size();

/*
** Kann Suchstring nicht komplett in der Datei liegen?
** => Nicht gefunden
*/
  if(pos+ssize>fsize)
    return(CFileArray::npos);

  streampos x;
  long y;

/*
** Suche nach String in Datei
*/
  for(x=pos,y=0;(x<=fsize-ssize)&&(y<ssize);x+=1,y++)
    if(search[y]!=(file[x])) {x-=y; y=-1;}

  if(y==ssize) return(x-y);
  else return(CFileArray::npos);
}
```

Listing 519: Die Funktion findFirstInFile (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

104 Wie kann bei der Textsuche in einer Datei die Groß- und Kleinschreibung ignoriert werden?

Dazu schreiben wir eine ähnliche Funktion wie `findFirstInFile` aus Rezept 103, nur dass die Zeichen beim Vergleich mit `tolower` »vorbehandelt« werden. Auch hier unterscheiden wir wieder zwischen der Version, die das `CFileArray`-Objekt selbst erstellt, und der Version, die ein bereits existierendes Objekt erwartet.

findFirstInFileCI

```
streampos findFirstInFileCI(const string &name,
                            const string &search,
                            streampos pos){
  try{

/*
** Datei öffnen und überladene Funktion aufrufen
*/
    CFileArray file(name,false);
    return(findFirstInFileCI(file,search,pos));
  }
  catch(...) {
    return(CFileArray::npos);
  }
}

//*********

streampos findFirstInFileCI(CFileArray &file,
                            const string &search,
                            streampos pos){

/*
** Größe der Datei und des
** Suchstrings ermitteln
*/
  streampos fsize=file.size();
  streampos ssize=search.size();

/*
** Kann Suchstring nicht komplett in der Datei liegen?
** => Nicht gefunden
*/
  if(pos+ssize>fsize)
    return(CFileArray::npos);

  streampos x;
  long y;
```

Listing 520: Die Funktion findFirstInFileCI

```
/*
** Suche nach String in Datei,
** Groß- Kleinschreibung wird ignoriert
*/
  for(x=pos,y=0;(x<=fsize-ssize)&&(y<ssize);x+=1,y++)
    if(tolower(search[y])!=tolower(file[x])) {x-=y; y=-1;}

  if(y==ssize) return(x-y);
  else return(CFileArray::npos);
}
```

Listing 520: Die Funktion findFirstInFileCI (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

105 Wie können binäre Daten in einer Datei gesucht werden?

Um binäre Daten in einer Datei suchen zu können, müssen die Daten, nach denen gesucht werden soll, zunächst in eine lineare, byteweise Form gebracht werden. Dabei werden aus Gründen der Effizienz nur Container mit wahlfreiem Zugriff eingesetzt: `vector` und `deque`.

Wir überladen dazu die zwei Funktionen `findFirstInFile` aus Rezept 103 für die beiden Container:

findFirstInFile für vector

```
streampos findFirstInFile(const string &name,
                          const vector<char> &search,
                          streampos pos){
  try{

/*
** Datei öffnen und überladene Funktion aufrufen
*/
    CFileArray file(name,false);
    return(findFirstInFile(file,search,pos));
  }
```

Listing 521: findFirstInFile für vector

```
  catch(...) {
    return(CFileArray::npos);
  }
}

//*********

streampos findFirstInFile(CFileArray &file,
                          const vector<char> &search,
                          streampos pos){
  try{

/*
** Größe der Datei und des
** Suchstrings ermitteln
*/
    streampos fsize=file.size();
    streampos ssize=search.size();

/*
** Kann Suchstring nicht komplett in der Datei liegen?
** => Nicht gefunden
*/
    if(pos+ssize>fsize)
      return(CFileArray::npos);

    streampos x;
    long y;

/*
** Suche in der Datei nach den Daten im Vektor
*/
    for(x=pos,y=0;(x<=fsize-ssize)&&(y<ssize);x+=1,y++)
      if(search[y]!=(file[x])) {x-=y; y=-1;}

    if(y==ssize) return(x-y);
    else return(CFileArray::npos);
  }
  catch(...) {
    return(CFileArray::npos);
```

Listing 521: findFirstInFile für vector (Forts.)

```
  }
}
```

Listing 521: findFirstInFile für vector (Forts.)

findFirstInFile für deque

```
streampos findFirstInFile(const string &name,
                          const deque<char> &search,
                          streampos pos){
  try{

/*
** Datei öffnen und überladene Funktion aufrufen
*/
    CFileArray file(name,false);
    return(findFirstInFile(file,search,pos));
  }
  catch(...) {
    return(CFileArray::npos);
  }
}

//*********

streampos findFirstInFile(CFileArray &file,
                          const deque<char> &search,
                          streampos pos){
  try{

/*
** Größe der Datei und des
** Suchstrings ermitteln
*/
    streampos fsize=file.size();
    streampos ssize=search.size();

/*
** Kann Suchstring nicht komplett in der Datei liegen?
** => Nicht gefunden
```

Listing 522: findFirstInFile für deque

```
*/
    if(pos+ssize>fsize)
      return(CFileArray::npos);

    streampos x;
    long y;

/*
** Sucht in der Datei nach den Daten in der Deque
*/
    for(x=pos,y=0;(x<=fsize-ssize)&&(y<ssize);x+=1,y++)
      if(search[y]!=(file[x])) {x-=y; y=-1;}

    if(y==ssize) return(x-y);
    else return(CFileArray::npos);
  }
  catch(...) {
    return(CFileArray::npos);
  }
}
```

Listing 522: findFirstInFile für deque (Forts.)

Die Funktionen finden Sie auf der CD in den `FileFunctions`-Dateien.

106 Wie kann mit binären Datei-Strömen gearbeitet werden?

Wenn auf normale Datenströme die Operatoren << oder >> angewendet werden, dann wird das Objekt immer in Zeichenform in den Strom geschrieben, beziehungsweise aus dem Strom gelesen.

Wenn dies häufig auch sehr praktisch ist, kann es manchmal auch wünschenswert sein, ein Objekt binär abzuspeichern. Dazu muss dann explizit eine binäre Datei geöffnet und das Objekt mit `read` oder `write` aus der Datei gelesen bzw. in die Datei geschrieben werden.

Wir wollen an dieser Stelle zwei eigene Klassen schreiben, die es uns ermöglichen werden, über die <<- und >>-Operatoren Objekte binär zu speichern und zu laden.

CBinaryOStream

Die Klasse `CBinaryOStream` wird alle Operatoren zum Speichern von Objekten beinhalten. Sie besitzt keine Attribute, sondern nur öffentliche Methoden. Die Stromspezifischen Methoden werden als abstrakt (rein-virtuell) definiert, um Unterklassen zu einer eigenen Implementierung zu zwingen.

Abstrakte Methoden

```
virtual void write(const void *mem, std::streamsize size)=0;
virtual bool is_open()=0;
virtual void close()=0;
virtual bool fail()=0;
```

Listing 523: Die abstrakten Methoden von CBinaryOStream

operator<< für elementare Datentypen

Für alle elementaren Datentypen wird nun der Operator << überladen, der alle Speicheroperationen auf Aufrufe der `write`-Methode reduziert.

```
CBinaryOStream &operator<<(char v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(short v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(int v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(long v) {
  write(&v, sizeof(v));
  return(*this);
}
```

Listing 524: operator<< für die elementaren Datentypen

```
CBinaryOStream &operator<<(unsigned short v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(unsigned int v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(unsigned long v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(float v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(double v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(long double v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(bool v) {
  write(&v, sizeof(v));
  return(*this);
}

CBinaryOStream &operator<<(const void *v) {
  write(&v, sizeof(v));
  return(*this);
}
```

Listing 524: operator<< für die elementaren Datentypen (Forts.)

operator<< für Strings

Dazu kommen noch die Varianten für C-Strings und STL-Strings. Weil die Länge zu Beginn bereits bekannt sein muss, wird beim Speichern nicht mehr mit der Endekennung gearbeitet, sondern die Länge explizit gespeichert:

```
CBinaryOStream &operator<<(const char *v) {
  size_t len=strlen(v);
  *this<<len;
  write(v, static_cast<std::streamsize>(len));
  return(*this);
}

CBinaryOStream &operator<<(const std::string &v) {
  size_t len=v.size();
  *this<<len;
  write(v.c_str(), static_cast<std::streamsize>(len));
  return(*this);
}
```

Listing 525: operator<< für Strings

operator<< für STL-Container

Damit auch alle in STL-Containern befindliche Objekte gespeichert werden können, wird `operator<<` auch für die Container überladen. Die Methoden sind als Templates gehalten, damit die Stream-Operatoren jeden beliebigen Datentyp im Container abdecken.

```
template<typename TypeA, typename TypeB>
CBinaryOStream &operator<<(const std::pair<TypeA,TypeB> &v) {
  *this << v.first << v.second;
  return(*this);
}

template<typename Type>
CBinaryOStream &operator<<(const std::vector<Type> &v) {
  *this<<v.size();
  for(vector<Type>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
```

Listing 526: operator<< für STL-Container

```
    *this<<*iter;
  return(*this);
}

template<typename Type>
CBinaryOStream &operator<<(const std::deque<Type> &v) {
  *this<<v.size();
  for(deque<Type>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
    *this<<*iter;
  return(*this);
}

template<typename Type>
CBinaryOStream &operator<<(const std::list<Type> &v) {
  *this<<v.size();
  for(list<Type>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
    *this<<*iter;
  return(*this);
}

template<typename Type>
CBinaryOStream &operator<<(const std::set<Type> &v) {
  *this<<v.size();
  for(set<Type>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
    *this<<*iter;
  return(*this);
}

template<typename Type>
CBinaryOStream &operator<<(const std::multiset<Type> &v) {
  *this<<v.size();
  for(multiset<Type>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
    *this<<*iter;
```

Listing 526: operator<< für STL-Container (Forts.)

```
  return(*this);
}

template<typename KType, typename DType>
CBinaryOStream &operator<<(const std::map<KType, DType> &v) {
  *this<<v.size();
  for(map<KType, DType>::const_iterator iter=v.begin();
      iter!=v.end();
      ++iter)
    *this<<*iter;
  return(*this);
}
```

Listing 526: operator<< für STL-Container (Forts.)

Die Klasse `CBinaryOStream` finden Sie auf der CD in der `CBinaryStream.h`-Datei.

CBinaryIStream

Das Gegenstück zu `CBinaryOStream` bildet die Klasse `CBinaryIStream`. Auch sie besitzt keine Attribute und nur öffentliche Methoden, von denen die Strom-spezifischen abstrakt sind:

Abstrakte Methoden

```
virtual void read(void *mem, std::streamsize size)=0;
virtual bool is_open()=0;
virtual void close()=0;
virtual bool fail()=0;
virtual bool eof()=0;
```

Listing 527: Die abstrakten Methoden von CBinaryIStream

operator>> für elementare Datentypen

Damit die elementaren Datentypen wieder aus einem Strom gelesen werden können, wird für sie `operator>>` überladen:

```
CBinaryIStream &operator>>(char &v) {
  read(&v, sizeof(v));
```

Listing 528: operator>> für die elementaren Datentypen

```
  return(*this);
}

CBinaryIStream &operator>>(short &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(int &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(long &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(unsigned short &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(unsigned int &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(unsigned long &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(float &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(double &v) {
  read(&v, sizeof(v));
  return(*this);
```

Listing 528: operator>> für die elementaren Datentypen (Forts.)

```
}

CBinaryIStream &operator>>(long double &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(bool &v) {
  read(&v, sizeof(v));
  return(*this);
}

CBinaryIStream &operator>>(const void *v) {
  read(&v, sizeof(v));
  return(*this);
}
```

Listing 528: operator>> für die elementaren Datentypen (Forts.)

operator>> für Strings

Im Folgenden sind die `operator>>`-Methoden für C-Strings und STL-Strings aufgeführt.

Um STL-Strings einlesen zu können, muss temporär ein Puffer reserviert werden, in den die Zeichenfolge eingelesen wird. Wie groß der Puffer sein muss, teilt uns die im Strom gespeicherte String-Größe mit.

```
CBinaryIStream &operator>>(char *v) {
  size_t len;
  *this>>len;
  read(v, static_cast<std::streamsize>(len));
  v[len]=0;
  return(*this);
}

CBinaryIStream &operator>>(std::string &v) {
  size_t len;
  *this>>len;
  char *buff=new char[len+1];
  read(buff, static_cast<std::streamsize>(len));
```

Listing 529: operator>> für Strings

```
  buff[len]=0;
  v=buff;
  delete(buff);
  return(*this);
}
```

Listing 529: operator>> für Strings (Forts.)

operator>> für STL-Container

Zum Schluss werden noch die Varianten für die STL-Container aufgeführt:

```
template<typename TypeA, typename TypeB>
CBinaryIStream &operator>>(std::pair<TypeA,TypeB> &v) {
  *this >> v.first >> v.second;
  return(*this);
}

template<typename Type>
CBinaryIStream &operator>>(std::vector<Type> &v) {
  v.clear();
  size_t size;
  *this>>size;
  Type input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.push_back(input);
  }
  return(*this);
}

template<typename Type>
CBinaryIStream &operator>>(std::deque<Type> &v) {
  v.clear();
  size_t size;
  *this>>size;
  Type input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.push_back(input);
  }
```

Listing 530: operator>> für STL-Container

```
  return(*this);
}

template<typename Type>
CBinaryIStream &operator>>(std::list<Type> &v) {
  v.clear();
  size_t size;
  *this>>size;
  Type input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.push_back(input);
  }
  return(*this);
}

template<typename Type>
CBinaryIStream &operator>>(std::set<Type> &v) {
  v.clear();
  size_t size;
  *this>>size;
  Type input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.insert(input);
  }
  return(*this);
}

template<typename Type>
CBinaryIStream &operator>>(std::multiset<Type> &v) {
  v.clear();
  size_t size;
  *this>>size;
  Type input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.insert(input);
  }
  return(*this);
}
```

Listing 530: operator>> für STL-Container (Forts.)

```
template<typename KType, typename DType>
CBinaryIStream &operator>>(std::map<KType, DType> &v) {
  v.clear();
  size_t size;
  *this>>size;
  pair<KType, DType> input;
  for(size_t i=0; i<size; i++) {
    *this>>input;
    v.insert(input);
  }
  return(*this);
}
```

Listing 530: operator>> für STL-Container (Forts.)

Die Klasse `CBinaryIStream` finden Sie auf der CD in der `CBinaryStream.h`-Datei.

CBinaryOFStream

Um die abstrakten Klassen `CBinaryOStream` und `CBinaryIStream` nutzen zu können, müssen wir konkrete Klassen von ihnen ableiten. Zunächst wollen wir uns die Möglichkeit schaffen, Daten in Dateien zu speichern und sie aus ihnen zu laden. Dazu implementieren wir die Klasse `CBinaryOFStream`. Sie wird von `CBinaryOStream` öffentlich abgeleitet und überschreibt ihre abstrakten Methoden.

Klassendefinition

```
class CBinaryOFStream : public CBinaryOStream {
private:
  std::ofstream m_stream;
};
```

Listing 531: Die Klassendefinition von CBinaryOFStream

Als Attribut besitzt die Klasse nur die mit ihr verknüpfte binäre Datei.

Konstruktor

```
CBinaryOFStream(const std::string &name) {
  m_stream.open(name.c_str(),std::ios::out|std::ios::binary);
}
```

Listing 532: Der Konstruktor von CBinaryOFStream

Der Konstruktor öffnet eine binäre Datei zum Schreiben.

write

```
void write(const void *mem, std::streamsize size) {
  m_stream.write(reinterpret_cast<const char*>(mem), size);
}
```

Listing 533: Die Methode write

`write` schreibt `size` Bytes der ab der Speicherposition `mem` liegenden Daten in die Datei.

Zugriffsmethoden

Die Zugriffsmethoden überschreiben die in der abstrakten Klasse `CBinaryOFStream` geforderten Methoden:

```
bool is_open() {
  return(m_stream.is_open());
}

void close() {
  m_stream.close();
}

bool fail() {
  return(m_stream.fail());
}
```

Listing 534: Die Zugriffsmethoden von CBinaryOFStream

Die Klasse `CBinaryOFStream` finden Sie auf der CD in der `CBinaryFStream.h`-Datei.

CBinaryIFStream

Um die mit `CBinaryOFStream` abgespeicherten Daten wieder laden zu können, implementieren wir die Klasse `CBinaryIFStream`, die von `CBinaryIStream` öffentlich abgeleitet wird:

Klassendefinition

```
class CBinaryIFStream : public CBinaryIStream {
private:
  std::ifstream m_stream;
};
```

Listing 535: Die Klassendefinition von CBinary

Konstruktor

Der Konstruktor ist ähnlich dem aus `CBinaryOFStream` aufgebaut. Er öffnet eine binäre Datei zum Lesen.

```
CBinaryIFStream(const std::string &name) {
  m_stream.open(name.c_str(),std::ios::in|std::ios::binary);
}
```

Listing 536: Der Konstruktor von CBinaryIFStream

read

`read` liest `size` Bytes aus der Datei und legt sie ab der Speicherposition `mem` im Speicher ab.

```
void read(void *mem, std::streamsize size) {
  m_stream.read(reinterpret_cast<char*>(mem), size);
}
```

Listing 537: Die Methode read

Zugriffsmethoden

Die Zugriffsmethoden überschreiben ihre abstrakten Verwandten und stellen damit einige der Streamabfragen zur Verfügung.

```
bool is_open() {
  return(m_stream.is_open());
}

void close() {
  m_stream.close();
}

bool fail() {
  return(m_stream.fail());
}

bool eof() {
  return(m_stream.eof());
}
```

Listing 538: Die Zugriffsmethoden von CBinaryIFStream

Die Klasse `CBinaryIFStream` finden Sie auf der CD in der `CBinaryFStream.h`-Datei.

Im Anschluss sehen Sie ein Beispiel, welches den Benutzer nach einem Verzeichnis fragt, dieses komplett einliest, abspeichert, anschließend wieder hineinlädt und als HTML-Dokument wieder abspeichert.

In der Praxis eine unsinnige Vorgehensweise, demonstriert es hier jedoch schön das Zusammenspiel der einzelnen Klassen und Funktionen.

```
#include "FileFunctions.h"
#include "HTMLFunctions.h"
#include "CBinaryFStream.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {

/*
** Benutzer nach Verzeichnisnamen fragen
*/
  cout << "Verzeichnis:";
```

Listing 539: Ein Beispiel zu CBinaryFStream

```
  string dir;
  getString(dir,200);

/*
** Komplettes Verzeichnis wird in filetree geladen
*/
  CFileInfo filetree;
  cout << "FileInfo erstellen..." << endl;
  if(getCompleteDir(dir,filetree)) {

/*
** Erzeugte Struktur wird mit CBinaryOFStream in die
** Datei TestStream.dat gespeichert
*/
    cout << "Stream speichern..." << endl;
    CBinaryOFStream ostream("TestStream.dat");
    ostream << filetree;
    ostream.close();

/*
** Eine neue Struktur loadedtree wird angelegt und
** die Daten von TestStream.dat in sie hineingeladen
*/
    cout << "Stream laden..." << endl;
    CFileInfo loadedtree;
    CBinaryIFStream istream("TestStream.dat");
    istream >> loadedtree;

/*
** Die Daten aus loadedtree werden in einen HTML-String
** umgewandelt
*/
    cout << "HTML-Dokument erstellen..." << endl;
    string htmltree=fileInfoToHtml(loadedtree,true,25,2);

/*
** Der HTML-String wird unter TestStream.html abgespeichert
*/
    cout << "HTML-Dokument speichern..." << endl;
    saveStringAsHtmlFile(htmltree,"TestStream.html");
    cout << "Fertig!" << endl;
```

Listing 539: Ein Beispiel zu CBinaryFStream (Forts.)

```
  }
  else
    cout << "ungueltiges Verzeichnis!" << endl;
}
```

Listing 539: Ein Beispiel zu CBinaryFStream (Forts.)

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0627.cpp*.

107 Wie kann auf Dateien wahlfrei mit Iteratoren zugegriffen werden?

Unter C++ besitzen Dateiströme bereits Iteratoren, allerdings mit der Einschränkung, dass sie nur auf Testebene arbeiten und im Normalfall aus Input- bzw. Output-Iteratoren bestehen, die nur unidirektional über einen Datenbereich iterieren können.

Selbst die `CBinaryStream`-Klassen aus Rezept 106 erlaubt uns nur einen unidirektionalen Zugriff.

Die Klasse `CFileArray` aus Rezept 102 bietet uns zwar die Möglichkeit auf Text- oder Byteebene wahlfrei über den Index-Operator auf eine Datei zuzugreifen, aber wahlfreien Zugriff über Iteratoren gab es auch dort nicht.

Der einfachste Ansatz liegt damit darin, die `CFileArray`-Klasse um einen Random-Access-Iterator zu erweitern.

Klassendefinition

Die Klasse `iterator` besitzt zwei Attribute: eine Referenz auf das dazugehörige `CFileArray`-Objekt und die aktuelle Position des Iterators in der Datei. `iterator` wird von der Klasse `std::iterator` abgeleitet, um sich in die STL-übliche Iteratoren-Hierarchie einzugliedern.

```
class iterator;
friend class CFileArray::iterator;
class iterator
: public std::iterator<std::random_access_iterator_tag, char> {
private:
  friend class CFileArray;
```

Listing 540: Die Klassendefinition vonCFileArray::iterator

```
  std::streampos m_index;
  CFileArray &m_fileArray;
};
```

Listing 540: Die Klassendefinition vonCFileArray::iterator (Forts.)

Durch die Angabe von `random_access_iterator_tag` ist festgeschrieben, welche Operatoren unser Iterator besitzen muss, nämlich die eines Random-Access-Iterators, und das sind ++, `--`, ==, `!=`, <, >, <=, >=, +, –, +=, `-=`, *, `->` und `[]`.

Konstruktoren

Es existieren zwei Konstruktoren: Der Kopier-Konstruktor und ein Konstruktor, der aus einer `CFileArray`-Referenz und einem Index einen Iterator erzeugt.

Einen Standard-Konstruktor gibt es nicht, weil der Iterator immer an ein `CFileArray`-Objekt gebunden sein muss.

```
iterator(CFileArray &d, std::streampos i=0)
  : m_fileArray(d), m_index(i)
{}

iterator(iterator &i)
  : m_fileArray(i.m_fileArray), m_index(i.m_index)
{}
```

Listing 541: Die Konstruktoren von iterator

Zuweisungsoperator

```
iterator &operator=(iterator &i) {
  m_fileArray=i.m_fileArray;
  m_index=i.m_index;
  return(*this);
}
```

Listing 542: Der Zuweisungsoperator von iterator

Inkrement und Dekrement

Die für Iteratoren fast schon wichtigsten Operatoren werden für unseren Iterator ebenfalls überladen, und zwar sowohl für die Prä- als auch für die Post-Schreibweise.

```
iterator &operator++() {
  if(m_index<m_fileArray.size())
    m_index+=1;
  return(*this);
}

iterator &operator--() {
  if(m_index>0)
    m_index-=1;
  return(*this);
}

iterator operator++(int) {
  iterator tmp=*this;
  if(m_index<m_fileArray.size())
    m_index+=1;
  return(tmp);
}

iterator operator--(int) {
  iterator tmp=*this;
  if(m_index>0)
    m_index-=1;
  return(tmp);
}
```

Listing 543: In- und Dekrement-Operatoren in der Prä- und Post-Schreibweise

Rechenoperatoren

Als Rechenoperatoren stehen Addition und Subtraktion als normale und als Zuweisungsoperatoren zur Verfügung. Dabei können Objekte vom Typ `iterator` und vom Typ `streampos` benutzt werden.

Bei der Addition oder Subtraktion zweier Objekte vom Typ `iterator` muss sichergestellt werden, dass beide mit demselben `CFileArray`-Objekt verknüpft sind. Sollte dies nicht der Fall sein, wird eine Ausnahme vom Typ `CFileArray::file_error` geworfen.

```
iterator operator+(std::streampos o) {
  std::streampos tmp=m_index+o;
  if(tmp>m_fileArray.m_size)
    tmp=m_fileArray.m_size;
  return(iterator(m_fileArray,tmp));
}

iterator operator-(std::streampos o) {
  std::streampos tmp;
  if(m_index<o)
    tmp=0;
  else
    tmp=m_index-o;
  return(iterator(m_fileArray,tmp));
}

iterator operator+(iterator &i) {
  if(&m_fileArray!=&i.m_fileArray)
    throw
    CFileArray::file_error("CFileArray::iterator::operator+()");
  std::streampos tmp=m_index+i.m_index;
  if(tmp>m_fileArray.m_size)
    tmp=m_fileArray.m_size;
  return(iterator(m_fileArray,tmp));
}

iterator operator-(iterator &i) {
  if(&m_fileArray!=&i.m_fileArray)
    throw
    CFileArray::file_error("CFileArray::iterator::operator-()");
  std::streampos tmp;
  if(m_index<i.m_index)
    tmp=0;
  else
    tmp=m_index-i.m_index;
  return(iterator(m_fileArray,tmp));
}

iterator &operator+=(std::streampos o) {
  m_index+=o;
  if(m_index>m_fileArray.m_size)
```

Listing 544: Die Rechenoperatoren von iterator

```
    m_index=m_fileArray.m_size;
  return(*this);
}

iterator &operator-=(std::streampos o) {
  if(m_index<o)
    m_index=0;
  else
    m_index-=o;
  return(*this);
}

iterator &operator+=(iterator &i) {
  if(&m_fileArray!=&i.m_fileArray)
    throw
    CFileArray::file_error("CFileArray::iterator::operator+=()");
  m_index+=i.m_index;
  if(m_index>m_fileArray.m_size)
    m_index=m_fileArray.m_size;
  return(*this);
}

iterator &operator-=(iterator &i) {
  if(&m_fileArray!=&i.m_fileArray)
    throw
    CFileArray::file_error("CFileArray::iterator::operator-=()");
  if(m_index<i.m_index)
    m_index=0;
  else
    m_index-=i.m_index;
  return(*this);
}
```

Listing 544: Die Rechenoperatoren von iterator (Forts.)

Vergleichsoperatoren

Ein Vergleich zweier `CFileArray::iterator`-Objekte ist nur dann sinnvoll, wenn beide mit demselben CFileArray-Objekt verknüpft sind.

Sind die beiden Iteratoren unterschiedlichen `CFileArray`-Objekten zugeordnet, liefert der Vergleich in jedem Fall `false`.

```
bool operator!=(iterator &i) {
  return((m_index!=i.m_index)&&(&m_fileArray==&i.m_fileArray));
}

bool operator==(iterator &i) {
  return((m_index==i.m_index)&&(&m_fileArray==&i.m_fileArray));
}

bool operator<(iterator &i) {
  return((m_index<i.m_index)&&(&m_fileArray==&i.m_fileArray));
}

bool operator>(iterator &i) {
  return((m_index>i.m_index)&&(&m_fileArray==&i.m_fileArray));
}

bool operator<=(iterator &i) {
  return((m_index<=i.m_index)&&(&m_fileArray==&i.m_fileArray));
}

bool operator>=(iterator &i) {
  return((m_index>=i.m_index)&&(&m_fileArray==&i.m_fileArray));
}
```

Listing 545: Die Vergleichsoperatoren von iterator

Index-Operator

Der Index-Operator liefert über `CFileArray` ein `CProxy`-Objekt zurück. Auf diese Weise ist sichergestellt, dass eine Veränderung der Daten über den Iterator wieder in die Datei zurück gespeichert wird.

```
CProxy operator[](std::streampos p) {
  return(m_fileArray[p]);
}
```

Listing 546: Der Index-Operator von iterator

operator*

`operator*` liefert ein `CProxy`-Objekt vom Zeichen am Index des Iterators zurück.

```
CProxy operator*() {
  return(m_fileArray[m_index]);
}
```

Listing 547: operator von iterator*

operator->

Weil unser Iterator nur auf ein `char`-Element verweist und nicht auf eine Struktur, macht ein Überladen der `operator->`-Methode eigentlich keinen Sinn.

Sie ist aber für einen Random-Accessoperator vorgeschrieben, deswegen ist sie bei uns von der Funktionalität her identisch mit der `operator*`-Methode.

```
CProxy operator->() {
  return(m_fileArray[m_index]);
}
```

Listing 548: operator-> von iterator

Die Klasse `iterator` finden Sie eingebettet in die Klasse `CFileArray` auf der CD in den `CFileArray`-Dateien.

108 Wie können Objekte byteweise in Container geschrieben werden?

Eine Frage auf diese Antwort ist insbesondere dann interessant, wenn beispielsweise in einer Datei nach Objekten gesucht werden soll. Liegt das zu suchende Objekt byteweise vor, dann kann nach seinem Abbild in der Datei gesucht werden.

Die meiste Arbeit wurde bereits mit den binären Strömen aus Rezept 106 erledigt.

Wir werden im Folgenden von den Klassen `CBinaryOStream` und `CBinaryIStream` ableiten, um die entsprechende Funktionalität zu implementieren. Auf diese Weise arbeiten die neuen Klassen problemlos mit all den anderen Klassen zusammen, die für binäre Ströme Stream-Operatoren zur Verfügung stellen.

CBinaryOContainerStream

Diese Klasse wird es uns ermöglichen Objekte mit Hilfe ihrer Stream-Operatoren in einen Container zu schreiben.

Wir leiten dazu von der Klasse `CBinaryOStream` ab.

Um ein Problem mit dem verwendeten Container mitteilen zu können, wird das Attribut `m_failed` eingeführt.

Klassendefinition

```
template<typename Type>
class CBinaryOContainerStream : public CBinaryOStream {
private:
  Type &m_container;
  bool m_failed;
};
```

Listing 549: Die Klassendefinition von CBinaryOContainerStream

Konstruktor

```
CBinaryOContainerStream(Type &container)
  : m_container(container), m_failed(false)
{}
```

Listing 550: Der Konstruktor von CBinaryOContainerStream

write

`write` schreibt ein Objekt byteweise in den Container. Sollten dabei mit dem Container Probleme auftreten, so wird `m_failed` auf `true` gesetzt.

```
void write(const void *mem, std::streamsize size) {
  try {
    const char *ptr=reinterpret_cast<const char*>(mem);
    for(streamsize i=0; i<size; i+=1) {
      m_container.push_back(ptr[i]);
    }
  }
  catch(...) {
    m_failed=true;
  }
}
```

Listing 551: Die Methode write von CBinaryOContainerStream

Zugriffsmethoden

Obwohl einige der Zugriffsmethoden bei Containern keinen Sinn machen (wie zum Beispiel `close`), müssen sie implementiert werden, um den Status der abstrakten Klasse abzulegen.

```
bool is_open() {
  return(true);
}

void close()
{}

bool fail() {
  return(m_failed);
}
```

Listing 552: Die Zugriffsmethoden von CBinaryOContainerStream

Die Klasse `CBinaryOContainerStream` finden Sie auf der CD in der `CBinaryContainerStream.h`-Datei.

CBinaryIContainerStream

`CBinaryIContainerStream` bildet das Gegenstück zu `CBinaryOContainerStream`. Die Klasse bietet die Möglichkeit, Daten byteweise aus einem Container herauszulesen und zu einem Objekt zusammenzusetzen.

Sie wird von `CBinaryIStream` abgeleitet.

Klassendefinition

Bei `CBinaryIContainerStream` ist ein weiteres Attribut notwendig: `m_pos` ist ein Iterator, der beim Auslesen aus dem Container über seine Daten läuft. Mit ihm lässt sich später auch das von `CBinaryIStream` geforderte `eof` realisieren.

```
template<typename Type>
class CBinaryIContainerStream : public CBinaryIStream {
private:
  const Type &m_container;
  bool m_failed;
```

Listing 553: Die Klassendefinition von CBinaryIContainerStream

```
  Type::const_iterator m_pos;
};
```

Listing 553: Die Klassendefinition von CBinaryIContainerStream (Forts.)

Konstruktor

Der Konstruktor setzt die Position `m_pos` auf dem Anfang des Containers.

```
CBinaryIContainerStream(Type &container)
  : m_container(container),
    m_failed(false),
    m_pos(container.begin())
{}
```

Listing 554: Der Konstruktor von CBinaryIContainerStream

read

`read` liest über den Iterator `m_pos` des Containers eine entsprechende Anzahl von Bytes in den Speicherbereich des Objekts.

```
void read(void *mem, std::streamsize size) {
  try {
    char *ptr=reinterpret_cast<char*>(mem);
    for(streamsize i=0; i<size; i+=1) {
      ptr[i]=*(m_pos++);
    }
  }
  catch(...) {
    m_failed=true;
  }
}
```

Listing 555: Die Methode read von CBinaryIContainerStream

Zugriffsmethoden

```
bool is_open() {
  return(true);
}

void close() {
}

bool fail() {
  return(m_failed);
}

bool eof() {
  return(m_pos==m_container.end());
}
```

Listing 556: Die Zugriffsmethoden von CBinaryIContainerStream

Die Klasse `CBinaryOContainerStream` finden Sie auf der CD in der `CBinaryContainerStream.h`-Datei.

Um die Funktionsweise der Container-Streams einmal zu veranschaulichen, wollen wir ein Objekt eines Datentyps, für den die Stream-Operatoren implementiert wurden, byteweise in einen Container schreiben und anschließend wieder auslesen.

Als Datentyp nehmen wir exemplarisch `CFileFilter` aus Rezept 97. Das Objekt `ff` wird angelegt und mit dem Muster »Ein Test« initialisiert.

Ein Vektor für `char`-Elemente namens `tvec` wird angelegt und mit dem `CBinaryOContainerStream`-Objekt `ocs` verknüpft.

Anschließend wird `ff` in den Stream hineingeschoben. Dann wird ein weiteres `CFileFilter`-Objekt angelegt und über den `CBinaryIContainerStream` `ics` wieder eingelesen.

Dazwischen wird noch zweimal auf `eof` geprüft.

```
CFileFilter ff("Ein Test");
vector<char> tvec;
CBinaryOContainerStream<vector<char> > ocs(tvec);
ocs << ff;
```

```
CFileFilter ff2;
CBinaryIContainerStream<vector<char> > ics(tvec);
cout << ics.eof() << endl;
ics >> ff2;

cout << ff2.getPattern() << endl;
cout << ics.eof() << endl;
```

In Kombination mit den `findFirstInFile`-Funktionen aus Rezept 105 kann nun auch nach vollständigen Objekten in einer Datei gesucht werden.

109 Wie können zwei Datei-Inhalte miteinander verglichen werden?

Dieses Problem lässt sich leicht unter Zuhilfenahme der Klasse `CFileArray` aus Rezept 102 lösen.

Die Funktion `compareFiles` bekommt zwei Dateinamen übergeben. Sie öffnet die beiden Dateien als `CFileArray`-Objekte und vergleicht sie byteweise.

Die Funktion gibt entweder die erste Position zurück, an der sich die beiden Dateien unterscheiden, oder `CFileArray::npos` bei Gleichheit.

```
streampos compareFiles(const string &name1,const string &name2) {
  streampos pos;
  try {

/*
** Beide Dateien öffnen ohne neue Dateien anzulegen
** und kleinere Dateigröße als Ende verwenden.
*/
    CFileArray file1(name1,false), file2(name2, false);
    streampos end=min(file1.size(), file2.size());

/*
** Dateien zeichenweise vergleichen
*/
    for(pos=0; pos<end; pos+=1)
      if(file1[pos]!=file2[pos])
```

Listing 557: Die Funktion compareFiles

```
        return(pos);

/*
** Unterschiedliche Größe?
** => Erste nur noch in einer Datei befindliche Position
**    zurückgeben
*/
    if(file1.size()!=file2.size())
      return(end);

/*
** Bei Gleichheit CfileArray::npos zurückgeben
*/
    return(CFileArray::npos);
  }

/*
** Eventuelle Ausnahmen fangen und eigene werfen
*/
  catch(...) {
    throw CFileArray::file_error("compareFiles()");
  }
}
```

Listing 557: Die Funktion compareFiles (Forts.)

Die Funktion finden Sie auf der CD in den `FileFunctions`-Dateien.

110 Wie kann in Dateien nach Textmustern gesucht werden?

Für diese Lösung greifen wir auf den Texterkennungsautomaten aus Rezept 18 zurück.

findFirstPatternInFile

Schauen wir uns zunächst die erste Variante der Funktion an. Sie besitzt folgende Parameter:

- `file` – ein `CFileArray`-Objekt
- `search` – das Suchmuster für den Automaten

- `casensitivity` – entscheidet, ob Groß-/Kleinschreibung berücksichtigt wird (`true`) oder nicht (`false`) (Standard: `true`)
- `p` – Position, ab der gesucht werden soll (Standard: 0)

```
streampos findFirstPatternInFile(CFileArray &file,
                                 const string &search,
                                 bool casesensitivity,
                                 streampos p) {
  try {
    CAutomat a(search);
    if(casesensitivity)
      return(a.find(file,p));
    else
      return(a.findCI(file,p));
  }
  catch(...) {
    throw invalid_argument("findFirstPatternInFile()");
  }
}
```

Listing 558: findFirstPatternInFile für CFileArray-Objekte

Um die Funktion eleganter aufrufen zu können, überladen wir sie um eine weitere Variante, der anstelle eines `CFileArray`-Objekts der Dateiname übergeben wird.

Sie besitzt folgende Parameter:

- `name` – der Dateiname
- `search` – das Suchmuster für den Automaten
- `casensitivity` – entscheidet, ob Groß-/Kleinschreibung berücksichtigt wird (`true`) oder nicht (`false`) (Standard: `true`)
- `p` – Position, ab der gesucht werden soll (Standard: 0)

```
streampos findFirstPatternInFile(const string &name,
                                 const string &search,
                                 bool casesensitivity,
                                 streampos p) {
  try {
```

Listing 559: findFirstPatternInFile für Dateinamen

```
    CFileArray fa(name);
    return(findFirstPatternInFile(fa,search,casesensitivity, p));
  }
  catch(CFileArray::file_error) {
    throw CFileArray::file_error("findfirstPatternInFile()");
  }
}
```

Listing 559: findFirstPatternInFile für Dateinamen (Forts.)

Die Funktionen finden Sie auf der CD in den `FileFunctions`-Dateien.

111 Wie kann nach Dateien mit bestimmten Textinhalten gesucht werden?

Die Frage lässt sich am einfachsten beantworten, indem die Klasse `CFileFilter` aus Rezept 97 durch Ableiten erweitert wird.

CTextFileFilter

Die Klasse `CTextFileFilter` ermöglicht es als zusätzliche Funktionalität, einen Text, den die Datei enthalten muss, als Suchkriterium anzugeben.

Als Attribute besitzt sie den zu suchenden Text (`m_searchText`) und die Information, ob bei der Suche zwischen Groß- und Kleinschreibung unterschieden werden soll oder nicht (`m_sCaseSensitivity`).

Klassendefinition

```
class CTextFileFilter : public CFileFilter{
private:
  std::string m_searchText;
  bool m_sCaseSensitivity;
};
```

Listing 560: Die Klassendefinition von CTextFileFilter

Konstruktor

Der Konstruktor initialisiert die beiden Attribute der Klasse und ruft den Basisklassen-Konstruktor auf.

```
CTextFileFilter(std::string pattern="?*",
                std::string stext="",
                bool casesensi=false,
                bool scasesensi=false,
                FTYPE types=BOTH,
                CFileSize minsize=CFileSize(0,0),
                CFileSize maxsize=CFileSize(0xffffffff,0xffffffff),
                CMoment minmodi=CMoment(1,1,1,0,0,0,0),
                CMoment maxmodi=CMoment(3000,12,31,23,59,59,999),
                CMoment mincrea=CMoment(1,1,1,0,0,0,0),
                CMoment maxcrea=CMoment(3000,12,31,23,59,59,999))

: CFileFilter(pattern, casesensi, types,
              minsize, maxsize,
              minmodi, maxmodi,
              mincrea, maxcrea),
  m_searchText(stext),
  m_sCaseSensitivity(scasesensi)
{}
```

Listing 561: Der Konstruktor von CTextFileFilter

Zugriffsmethoden

```
void setSearchText(const std::string &p) {
  m_searchText=p;
}

void setSCaseSensitivity(bool cs) {
  m_sCaseSensitivity=cs;
}

std::string getSearchText() const {
  return(m_searchText);
}

bool getSCaseSensitivity() const {
  return(m_sCaseSensitivity);
}
```

Listing 562: Die Zugriffsmethoden von CtextFileFilter

match

Die Methode `match` überprüft, ob die entsprechende Datei den Kriterien genügt.

```
bool CTextFileFilter::match(const CFileInfo &fi) const {

/*
** Kriterien der Basisklasse prüfen
*/
  if(!CFileFilter::match(fi))
    return(false);

/*
** Dateityp nicht FILE?
** => Keine weitere Überprüfung
*/
  if(fi.m_type!=CFileInfo::FILE)
    return(false);

/*
** Datei nach Text durchsuchen. Je nach Berücksichtigung
** von Groß-/Kleinschreibung entsprechende Funktion aufrufen
*/
  try {
    if(m_sCaseSensitivity)
      return(findFirstInFile(fi.m_path+"/"+fi.m_name,m_searchText)!=
             CFileArray::npos);
    else
      return(findFirstInFileCI(fi.m_path+"/"+fi.m_name,m_searchText)!=
       CFileArray::npos);
  }
  catch(...) {
    throw CFileArray::file_error("CTextFileFilter::match()");
  }
}
```

Listing 563: match von CTextFileFilter

Stream-Operatoren

Die Klasse wird noch mit Stream-Operatoren ausgestattet, damit sie auch über die Stream-Klassen aus Rezept 106 gespeichert und geladen werden kann.

```
CBinaryOStream &operator<<(CBinaryOStream &os,
                           const CTextFileFilter &d) {
  os << static_cast<CFileFilter>(d);
  os << d.m_searchText;
  os << d.m_sCaseSensitivity;
  return(os);
}

CBinaryIStream &operator>>(CBinaryIStream &is, CTextFileFilter &d) {
  is >> static_cast<CFileFilter>(d);
  is >> d.m_searchText;
  is >> d.m_sCaseSensitivity;
  return(is);
}
```

Listing 564: Die Stream-Operatoren vonCTextFileFilter

Die Klasse `CTextFileFilter` finden Sie auf der CD in den `CTextFileFilter`-Dateien.

CPatternFileFilter

Für den Fall, dass in der Datei nicht nur ein starrer Text, sondern ein Muster gesucht werden soll, wollen wir die Klasse `CPatternFileFilter` von `CFileFilter` ableiten.

Als Attribute besitzt sie das Suchmuster (`m_searchPattern`), einen booleschen Wert für die Berücksichtigung der Groß- und Kleinschreibung (`m_sCaseSensitivity`) und den Automaten, der in der Datei nach dem Muster sucht (`m_searchAutomat`).

Klassendefinition

```
class CPatternFileFilter : public CFileFilter{
private:
  std::string m_searchPattern;
  bool m_sCaseSensitivity;
  CAutomat m_searchAutomat;
};
```

Listing 565: Die Klassendefinition von CPatternFileFilter

Konstruktor

```
CPatternFileFilter(std::string pattern="?*",
                   std::string spattern="?*",
                   bool casesensi=false,
                   bool scasesensi=false,
                   FTYPE types=BOTH,
                   CFileSize minsize=CFileSize(0,0),
                   CFileSize maxsize=CFileSize(0xffffffff,
                                               0xffffffff),
                   CMoment minmodi=CMoment(1,1,1,0,0,0,0),
                   CMoment maxmodi=CMoment(3000,12,31,23,59,59,999),
                   CMoment mincrea=CMoment(1,1,1,0,0,0,0),
                   CMoment maxcrea=CMoment(3000,12,31,23,59,59,999))

: CFileFilter(pattern, casesensi, types,
              minsize, maxsize,
              minmodi, maxmodi,
              mincrea, maxcrea),
  m_searchPattern(spattern),
  m_sCaseSensitivity(scasesensi),
  m_searchAutomat((scasesensi)?spattern:toLowerString(spattern))
{}
```

Listing 566: Der Konstruktor von CPatternFileFilter

Zugriffsmethoden

```
void setSearchPattern(const std::string &p) {
  m_searchPattern=p;
}

void setSCaseSensitivity(bool cs) {
  m_sCaseSensitivity=cs;
}

std::string getSearchPattern() const {
  return(m_searchPattern);
}
```

Listing 567: Die Zugriffsmethoden von CPatternFileFilter

```
bool getSCaseSensitivity() const {
  return(m_sCaseSensitivity);
}
```

Listing 567: Die Zugriffsmethoden von CPatternFileFilter (Forts.)

match

```
bool CPatternFileFilter::match(const CFileInfo &fi) const {

/*
** Kriterien der Basisklasse prüfen
*/
  if(!CFileFilter::match(fi))
    return(false);

/*
** Dateityp nicht FILE?
** => Keine weitere Überprüfung
*/
  if(fi.m_type!=CFileInfo::FILE)
    return(false);

/*
** Datei nach Text durchsuchen. Je nach Berücksichtigung
** von Groß-/Kleinschreibung entsprechende Methode aufrufen
*/
  try {
  CFileArray fa(fi.m_path+"/"+fi.m_name);

    if(m_sCaseSensitivity)
      return(m_searchAutomat.find(fa)!=CFileArray::npos);
    else
      return(m_searchAutomat.findCI(fa)!=CFileArray::npos);

  }
  catch(...) {
    throw CFileArray::file_error("CPatternFileFilter::match()");
  }
}
```

Listing 568: Die match-Methode von CPatternFileFilter

Stream-Operatoren

```
CBinaryOStream &operator<<(CBinaryOStream &os,
                           const CPatternFileFilter &d) {
  os << static_cast<CFileFilter>(d);
  os << d.m_searchPattern;
  os << d.m_sCaseSensitivity;
  return(os);
}

CBinaryIStream &operator>>(CBinaryIStream &is,
                           CPatternFileFilter &d) {
  is >> static_cast<CFileFilter>(d);
  is >> d.m_searchPattern;
  is >> d.m_sCaseSensitivity;
  d.m_searchAutomat.rebuild(d.m_searchPattern);
  return(is);
}
```

Die Klasse `CPatternFileFilter` finden Sie auf der CD in den `CPatternFileFilter`-Dateien.

112 Wie wird geprüft, ob eine Zahl gerade oder ungerade ist?

Eine Zahl ist genau dann gerade, wenn sie ohne Rest durch 2 teilbar ist. Wir schreiben dazu eine Template-Funktion, die diese Berechnung für jeden Datentyp vornimmt, der mit einem `%`-Operator ausgestattet ist.

```
template<typename Type>
bool isEven(Type v) {
  return(!(v%2));
}
```

Listing 569: Die Template-Funktion isEven

Die Funktion `isEven` finden Sie auf der CD in den `MathFunctions`-Dateien.

113 Wie wird geprüft, ob eine Zahl eine Primzahl ist?

Ein Zahl ist genau dann prim, wenn sie nur durch 1 und durch sich selbst teilbar ist. Dabei wird die 1 selbst nicht als Primzahl angesehen.

Die Funktion `isPrime` teilt eine Zahl n durch alle ungeraden Zahlen im Bereich von 3 bis Wurzel n. Gerade Zahlen müssen nicht geprüft werden, denn wenn eine Zahl durch eine gerade Zahl teilbar ist, dann ist sie auch durch zwei teilbar, und das wird gleich am Anfang der Funktion überprüft.

```
template<typename Type>
bool isPrime(Type v) {

/*
** Sonderfall: 1 ist keine Primzahl
*/
  if(v==1)
    return(false);
```

Listing 570: Die Template-Funktion isPrime

```
/*
** 2 ist eine Primzahl
*/
  if(v==2)
    return(true);

/*
** Durch 2 teilbar?
** => Keine Primzahl
*/
  if(!(v%2))
    return(false);

/*
** von 3 ab in Zweierschritten bis Wurzel(v) auf teilbar prüfen
*/
  Type end=static_cast<Type>(sqrt(v));
  for(Type i=3;i<=end; i+=2)
    if(!(v%i))
      return(false);
  return(true);
}
```

Listing 570: Die Template-Funktion isPrime (Forts.)

Die Funktion `isPrime` finden Sie auf der CD in den `MathFunctions`-Dateien.

114 Wie werden alle Primzahlen von 1 bis n ermittelt?

Alle Primzahlen in einem bestimmten Bereich zu ermitteln, erfordert eine etwas andere Vorgehensweise, als jede Zahl in diesem Bereich mit der Funktion `isPrime` aus Rezept 113 auf prim zu überprüfen.

Wir setzen dazu das Sieb des Erathostenes ein. Alle Zahlen, die potenziell prim sein könnten, beginnend mit 2, werden »aufgeschrieben«.

Dann wird die erste Zahl dieser Liste genommen und alle Vielfache von ihr aus der Liste gestrichen.

Anschließend wird die nächste Zahl der Liste genommen und wiederum alle Vielfache von ihr aus der Liste gestrichen.

Dies macht man so lange, bis die nächste Zahl der Liste größer die Wurzel von n ist. Alle übrig gebliebenen Zahlen sind Primzahlen.

Wir implementieren diesen Ansatz in einer Funktion `getPrimeNumbers`, die den Wert, bis zu dem nach Primzahlen gesucht werden soll und ein Vektor, der das Ergebnis übernimmt, übergeben werden.

```
void getPrimeNumbers(unsigned long v,
                     vector<unsigned long> &result) {
  unsigned long i;

/*
** Ein entsprechend großes Feld reservieren
** Und Endewert bestimmen
*/
  vector<unsigned long> field(v+1,0);
  unsigned long end=static_cast<unsigned long>(sqrt(v));

/*
** Alle ungeraden Zahlen als potenzielle
** Primzahlen eintragen
*/
  for(i=3; i<=v; i+=2)
    field[i]=i;

  i=3;
  do {

/*
** Alle Vielfachen von i können keine Primzahlen sein
** => Aus der Liste austragen
*/
    for(unsigned long x=i*2; x<=v; x+=i)
      field[x]=0;
    i+=2;

/*
** Nächsten Wert aus der Liste suchen oder abbrechen,
** falls Grenze erreicht
*/
    while((i<=end)&&(!field[i]))
```

Listing 571: Die Funktion getPrimeNumbers

```
      i+=2;
  } while(i<=end);

/*
** Alle ermittelten Primzahlen in den
** Ergebnisvektor übertragen
*/
  result.push_back(2);
  for(i=3; i<=v; i+=2)
    if(field[i])
      result.push_back(field[i]);
}
```

Listing 571: Die Funktion getPrimeNumbers (Forts.)

Die Funktion `getPrimeNumbers` finden Sie auf der CD in den `MathFunctions`-Dateien.

115 Wie wird der größte gemeinsame Teiler berechnet?

Wir berechnen den größten gemeinsamen Teiler nach Euklid.

Soll der größte gemeinsame Teiler zweier Zahlen a und b gefunden werden, so wird zunächst der Rest der Division von a durch b (a%b) ermittelt.

Ist dieser Rest 0, dann ist b der größte gemeinsame Teiler.

Ist der Rest nicht 0, dann ist der größte gemeinsame Teiler von a und b identisch mit dem größten gemeinsamen Teiler von b und dem vorher ermittelten Rest.

In der Funktion `ggT` sind diese Erkenntnisse verdichtet:

```
template<typename Type>
Type ggT(Type a, Type b) {
  Type c=a%b;
  if(!c)
    return(b);
  else
    return(ggT(b,c));
}
```

Listing 572: Die Funktion ggT

Die Funktion `ggT` finden Sie auf der CD in den `MathFunctions`-Dateien.

116 Wie wird das kleinste gemeinsame Vielfache berechnet?

Mit der Formel

kgV(a,b) * ggT(a,b) = a * b

ist das kleinste gemeinsame Vielfache leicht zu berechnen, zumal uns aus Rezept 115 bereits eine Funktion zur Berechnung des größten gemeinsamen Teilers zur Verfügung steht.

```
template<typename Type>
Type kgV(Type a, Type b) {
  return(a*b/ggT(a,b));
}
```

Listing 573: Die Funktion kgV

Die Funktion `kgV` finden Sie auf der CD in den `MathFunctions`-Dateien.

117 Wie wird die Fakultät einer Zahl berechnet?

Die Fakultät einer Zahl n ist das Produkt aller natürlichen Zahlen von 1 bis n und wird n! geschrieben. 6! ist damit 1*2*3*4*5*6=720.

Die Funktion `facul` ist die dazugehörige C++-Umsetzung:

```
template<typename Type>
Type facul(Type x) {
  if(x<0)
    throw std::out_of_range("facul()");
  Type f=1;
  while(x>1)
    f*=x--;
  return(f);
}
```

Listing 574: Die Funktion facul

Die Funktion `facul` finden Sie auf der CD in den `MathFunctions`-Dateien.

118 Wie wird die Fibonacci-Zahl von n berechnet?

Die Fibonacci-Zahl von n ist definiert als die Summe der Fibonacci-Zahlen von n-1 und n-2 mit der Besonderheit, dass die Fibonacci-Zahlen von 1 und 2 den Wert 1 haben.

Die rekursive Funktion `fibonacci` setzt die Rechenvorschrift in C++ um:

```
template<typename Type>
Type fibonacci(Type x) {
  if(x<1)
    throw std::out_of_range("fibonacci()");
  if((x<3))
    return(1);
  return( fibonacci(x-1) + fibonacci(x-2) );
}
```

Listing 575: Die Funktion fibonacci

Die Funktion `fibonacci` finden Sie auf der CD in den `MathFunctions`-Dateien.

119 Wie kann mit beliebig großen Ganzzahlen gearbeitet werden?

Nach der ANSI-Norm ist der Datentyp `long` bzw. `unsigned long` mit 32 Bits der größtmögliche Datentyp für Ganzzahlen im C++-Universum.

Manche Umgebungen bieten auch noch 64-Bit-Datentypen an, aber darüber hinaus findet sich meist nichts mehr.

Wir wollen hier eine Klasse `CHugeNumber` implementieren, die mit beliebig großen Ganzzahlen arbeiten kann.

Damit die Vorgehensweise verständlicher ist, werden wir die Effizienz etwas im Hintergrund lassen und die Daten möglichst greifbar verwalten. Die Klasse wird sich daher nicht für hochprofessionelle Berechnungen eignen (bei denen meist sowieso mit Fließkommazahlen gearbeitet wird), sondern ist eher für die »normalen« bestimmt.

Mit dem Verständnis der einzelnen Rechenschritte lassen sich jedoch leicht effizientere Klassen entwickeln.

CDigit

Zunächst entwerfen wir eine Klasse `CDigit`, die eine Ziffer unserer Zahl repräsentiert. Um die Effizienz zu steigern, würde man bei kommerziellen Anwendungen auf eine eigene Klasse für die Ziffern verzichten.

Klassendefinition

```
class CDigit {
public:
  digit_type m_digit;
};
```

Listing 576: Die Klassendefinition von CDigit

Das Attribut `m_digit` ist öffentlich, weil die Klasse im privaten Teil von `CHugeNumber` definiert wird und somit von außen ohnehin nicht zugänglich ist.

Da wir Ganzzahlen im Dezimalsystem verwalten möchten, liegt der Wertebereich von 0 bis 9. Der Datentyp `digit_type` wird in der Klasse `CHugeNumber` auf `char` gesetzt. Auch dies ist kein Feuerwerk der Effizienz, weil ein `char`-Element 8 Bit benötigt, man aber rein rechnerisch zur Kodierung dieses Wertebereichs nur knappe 3,4 Bits bräuchte. Durch die Wahl von `char` können die einzelnen Ziffern jedoch einfacher angesprochen werden und Manipulationen auf Bitebene bleiben erspart.

Konstruktoren

Die Konstruktoren bieten die Möglichkeit, ein `CDigit`-Objekt entweder aus einem `char`- oder einem `int`-Wert zu erzeugen.

```
CHugeNumber::CDigit::CDigit(int w) {
  if((w<0)||(w>9))
    throw out_of_range("CHugeNumber::CDigit::CDigit()");
  m_digit=w;
}

//*********

CHugeNumber::CDigit::CDigit(char w) {
  if((w<'0')||(w>'9'))
    throw out_of_range("CHugeNumber::CDigit::CDigit()");
```

Listing 577: Die Konstruktoren von CDigit

```
  m_digit=w-'0';
}
```

Listing 577: Die Konstruktoren von CDigit (Forts.)

Rechenoperatoren

Bei den Rechenoperatoren wurde etwas getan, von dem in sämtlichen Lehrbüchern abgeraten wird: Die Operatoren verhalten sich nicht so, wie sie es bei den elementaren Datentypen tun.

Dieser Bruch mit einer ansonsten unbedingt einzuhaltenden Regel ist insofern nicht weiter tragisch, als die Operatoren nur innerhalb von `CDigit` und `CHugeNumber` verfügbar sind und der Benutzer mit ihnen nie in Kontakt kommt.

Die Besonderheit der folgenden Zuweisungsoperatoren liegt darin, dass sie nicht wie gewöhnlich eine Referenz auf das eigene Objekt zurückliefern, sondern einen booleschen Wert, der mitteilt, ob es bei der Addition einen Überlauf bzw. bei der Subtraktion einen Unterlauf gegeben hat.

Bei einem Wertebereich von 0 bis 9 tritt ein Überlauf dann auf, wenn das Ergebnis der Addition größer als 9 ist oder das Ergebnis der Subtraktion kleiner 0.

Diese Mitteilung über einen Über-/Unterlauf ist wichtig, weil er sich auf die nächste Stelle der gesamten Zahl auswirkt. Addiere ich bei der Zahl 28 eine 3 auf die 8, dann ist die Ziffer 2 ebenfalls betroffen, weil bei der Addition ein Überlauf stattfindet.

```
bool CHugeNumber::CDigit::operator+=(int d) {
  m_digit+=d;

/*
** Überlauf?
** => Überlauf abziehen und zurückgeben
*/
  if(m_digit>9) {
    m_digit%=10;
    return(true);
  }
  else
    return(false);
}
```

Listing 578: Die Rechenoperatoren von CDigit

```
//*********

bool CHugeNumber::CDigit::operator+=(const CDigit &d) {
  m_digit+=d.m_digit;

/*
** Überlauf?
** => Überlauf abziehen und zurückgeben
*/
  if(m_digit>9) {
    m_digit%=10;
    return(true);
  }
  else
    return(false);
}

//*********

bool CHugeNumber::CDigit::operator-=(int d) {

/*
** Unterlauf?
** => Unterlauf hinzufügen und zurückgeben
*/
  if(m_digit>=d) {
    m_digit-=d;
    return(false);
  }
  else {
    m_digit=m_digit+10-d;
    return(true);
  }
}

//*********

bool CHugeNumber::CDigit::operator-=(const CDigit &d) {

/*
** Unterlauf?
```

Listing 578: Die Rechenoperatoren von CDigit (Forts.)

```
** => Unterlauf hinzufügen und zurückgeben
*/
  if(m_digit>=d.m_digit) {
    m_digit-=d.m_digit;
    return(false);
  }
  else {
    m_digit=m_digit+10-d.m_digit;
    return(true);
  }
}
```

Listing 578: Die Rechenoperatoren von CDigit (Forts.)

Vergleichsoperatoren

Um die Klasse `CDigit` zu komplettieren kommen nun die Vergleichsoperatoren, die sich ziemlich einfach gestalten.

```
bool CHugeNumber::CDigit::operator==(const CDigit &d) const{
  return(d.m_digit==m_digit);
}

//*********

bool CHugeNumber::CDigit::operator!=(const CDigit &d) const{
  return(d.m_digit!=m_digit);
}

//*********

bool CHugeNumber::CDigit::operator<(const CDigit &d) const{
  return(m_digit<d.m_digit);
}

//*********

bool CHugeNumber::CDigit::operator>(const CDigit &d) const{
  return(m_digit>d.m_digit);
}
```

Listing 579: Die Vergleichsoperatoren von CDigit

```
//*********

bool CHugeNumber::CDigit::operator<=(const CDigit &d) const{
  return(m_digit<=d.m_digit);
}

//*********

bool CHugeNumber::CDigit::operator>=(const CDigit &d) const{
  return(m_digit>=d.m_digit);
}
```

Listing 579: Die Vergleichsoperatoren von CDigit (Forts.)

CHugeNumber

Kommen wir nun zur Klasse `CHugeNumber`, die aus den einzelnen `CDigit`-Ziffern eine vollständige Zahl macht.

Die Klasse besitzt zwei Attribute, einen Container, der die einzelnen Ziffern aufnimmt, und einen `short`-Wert, der das Vorzeichen der Zahl speichert.

Klassendefinition

```
class CHugeNumber {
private:
  typedef char digit_type;

  class CDigit { /* ... */ };

public:
  typedef std::deque<CDigit> number_type;

private:
  number_type m_number;
  short m_sign;
};
```

Listing 580: Die Klassendefinition von CHugeNumber

Konstruktoren

Der erste Konstruktor erzeugt ein `CHugeNumber`-Objekt aus einem C++-String.

Dabei muss auf ein eventuelles Vorzeichen geachtet werden.

Die interne Struktur von `CHugeNumber`-Objekten ist so aufgebaut, dass die Ziffer mit der niedrigsten Wertigkeit am Anfang des Containers liegt. Bei einer in einem String abgelegten Zahl ist dies genau umgekehrt. Die Schleife, mit der die einzelnen Ziffern vom String in den Container des `CHugeNumber`-Objekts übertragen werden, arbeitet deswegen mit Reverse-Iteratoren.

```
CHugeNumber::CHugeNumber(const std::string &w) {
  string str;

/*
** Ein - am Anfang der Zahl?
** => m_sign auf -1 setzen und - ignorieren
*/
  if(w[0]=='-') {
    m_sign=-1;
    str=w.substr(1);
  }

/*
** Ein + am Anfang der Zahl?
** => m_sign auf 1 setzen und + ignorieren
*/
  else if(w[0]=='+') {
    m_sign=1;
    str=w.substr(1);
  }

/*
** Kein Vorzeichen?
** => m_sign auf 1 setzen
*/
  else {
    m_sign=1;
    str=w;
  }

/*
** Im String hat erste Ziffer größte Wertigkeit, in CHugeNumber
** erste Ziffer kleinste Wertigkeit
```

Listing 581: Der Konstruktor für Strings

```
** => String von hinten nach vorne übertragen
*/
  string::reverse_iterator i=str.rbegin();
  for(;i!=str.rend(); ++i)
    m_number.push_back(CDigit(*i));

/*
** Eventuell führende Nullen löschen
*/
  _shorten(m_number);
}
```

Listing 581: Der Konstruktor für Strings (Forts.)

Die Konstruktoren für die Ganzzahlen reduzieren das Problem auf Strings, indem sie die Zahlen in Strings umwandeln und dann die einzelnen Ziffern in `CDigit`-Objekte konvertieren.

```
CHugeNumber::CHugeNumber(long w) {

/*
** Vorzeichen ermitteln
*/
  m_sign=(w<0)?-1:1;

/*
** Vorzeichenlose Zahl in einen String umwandeln
** und mit Reverse-Iteratoren übertragen
*/
  string s=toString(labs(w));
  for(string::reverse_iterator i=s.rbegin(); i!=s.rend(); ++i)
    m_number.push_back(CDigit(*i));
}

//*********

CHugeNumber::CHugeNumber(unsigned long w) {

/*
** Unsigned-Werte sind immer positiv
```

Listing 582: Die Konstruktoren für Ganzzahlen

```
*/
  m_sign=1;

/*
** Zahl in einen String umwandeln
** und mit Reverse-Iteratoren übertragen
*/
  string s=toString(w);
  for(string::reverse_iterator i=s.rbegin(); i!=s.rend(); ++i)
    m_number.push_back(CDigit(*i));
}

//*********

CHugeNumber::CHugeNumber(int w) {

/*
** Vorzeichen ermitteln
*/
  m_sign=(w<0)?-1:1;

/*
** Vorzeichenlose Zahl in einen String umwandeln
** und mit Reverse-Iteratoren übertragen
*/
  string s=toString(abs(w));
  for(string::reverse_iterator i=s.rbegin(); i!=s.rend(); ++i)
    m_number.push_back(CDigit(*i));
}
```

Listing 582: Die Konstruktoren für Ganzzahlen (Forts.)

Ein Konstruktor, der ein `CHugeNumber`-Objekt aus den Einzelteilen eines anderen `CHugeNumber`-Objekts erzeugt, darf auch nicht fehlen.

```
CHugeNumber::CHugeNumber(const number_type &v, short sign)
: m_number(v), m_sign(sign)
{}
```

Listing 583: Ein Konstruktor für die Einzelteile von CHugeNumber

Zum Schluss noch ein Konstruktor, der speziell für die bessere Zusammenarbeit mit der Klasse `CFileSize` aus Rezept 84 implementiert wurde und ein `CHugeNumber`-Objekt aus einer 64-Bit-Zahl erzeugt, die in Form zweier vorzeichenloser 32-Bit-Werte vorliegt:

```
CHugeNumber::CHugeNumber(unsigned long low, unsigned long high) {
  *this=high*CHugeNumber("4294967296")+low;[a]
}
```

Listing 584: Ein Konstruktor für 6-Bit-Werte

a. $4294967296 = 2^{32}$

increase

Die private Methode `increase` erhöht eine bestimmte Ziffer der Zahl um 1 und gibt eventuelle Überträge an die nächsthöheren Zahlen weiter.

Diese Methode findet später Verwendung bei der Addition und den ++-Operatoren.

```
void CHugeNumber::increase(CHugeNumber::number_type &v, int p) {
  bool overflow;
  do {

/*
** Übertrag auf eine nicht existierende Ziffer?
** => Ziffer hinzufügen
*/
    if(p==v.size())
      v.push_back(0);

/*
** Ziffer um 1 erhöhen und Übertrag speichern
*/
    overflow=v[p++]+=1;

/*
** Übertrag vorhanden?
** => An nächste Ziffer weitergeben
*/
  } while(overflow);
}
```

Listing 585: Die private Methode increase

decrease

Die private Methode `decrease` vermindert eine bestimmte Ziffer der Zahl um 1 und gibt eventuelle Überträge an die nächsthöheren Zahlen weiter.

Zum Schluss werden entstandene führende Nullen (ein `decrease` bei 100 ergibt 099) gelöscht.

Diese Methode findet später Verwendung bei der Subtraktion und den `---`-Operatoren.

```
void CHugeNumber::decrease(CHugeNumber::number_type &v, int p) {
  bool overflow;
  do {

/*
** Ziffer um 1 vermindern und Übertrag speichern
*/
    overflow=v[p++]-=1;

/*
** Übertrag vorhanden?
** => An nächste Ziffer weitergeben
*/
  } while(overflow);

/*
** Eventuell entstandende führende Nullen kürzen
*/
  shorten(v);
}
```

Listing 586: Die private Methode decrease

add

Die private Methode `add` addiert zwei `number_type` Container und liefert das Ergebnis als Rückgabewert, wobei die Möglichkeit besteht, den zweiten Summanden zu verschieben. Dadurch sind Additionen wie in Abbildung 41 dargestellt möglich, die die Grundlage einer späteren Multiplikation bilden.

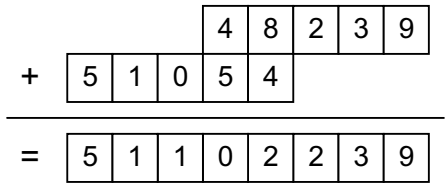


Abbildung 41: Eine Addition mit verschobenem zweiten Summanden

```
CHugeNumber::number_type
CHugeNumber::add(const CHugeNumber::number_type &v1,
                 const CHugeNumber::number_type &v2,
                 unsigned int p) {

  number_type r=v1;

/*
** Existiert Ziffer im ersten Summanden nicht?
** => Nullen hinzufügen
*/
  while(r.size()<=p)
    r.push_back(0);

/*
** Alle Ziffern werden addiert und
** ein Übertrag bei der nächsthöheren Ziffer berücksichtigt.
*/
  int overflow=(r[p++]+=v2[0]);

  for(unsigned int i=1; i<v2.size(); i++) {
    if(p==r.size())
      r.push_back(0);
    overflow=(r[p++]+=(v2[i].m_digit+overflow));
  }

/*
** Zweiter Summand komplett hinzuaddiert, aber noch Übertrag
** => increase erledigt den Rest
*/
  if(overflow)
    increase(r,p);
```

Listing 587: Die private Methode add

```
/*
** Eventuell zu viel angehängte Nullen werden gekürzt.
*/
  shorten(r);
  return(r);
}
```

Listing 587: Die private Methode add (Forts.)

sub

Die private Methode `sub` subtrahiert zwei `number_type`-Container voneinander und liefert das Ergebnis als Rückgabewert.

```
CHugeNumber::number_type
CHugeNumber::sub(const CHugeNumber::number_type &v1,
                 const CHugeNumber::number_type &v2) {

/*
** Invariante: Erster Operand muss größer sein als zweiter
*/
  if(compare(v1,v2)>0)
    throw out_of_range("CHugeNumber::_sub()");

  number_type r=v1;

/*
** Alle Ziffern werden subtrahiert und
** ein Übertrag bei der nächsthöheren Ziffer berücksichtigt.
*/
  int p=0;
  int overflow=(r[p++]-=v2[0]);

  for(unsigned int i=1; i<v2.size(); i++)
    overflow=(r[p++]-=(v2[i].m_digit+overflow));

/*
** Zweiter Operand komplett abgezogen, aber noch Übertrag
** => decrease erledigt den Rest
*/
  if(overflow)
```

Listing 588: die private Methode sub

```
    decrease(r,p);

/*
** Eventuell entstandene führende Nullen werden gekürzt.
*/
  shorten(r);
  return(r);
}
```

Listing 588: die private Methode sub (Forts.)

mul

Die Methode `mul` multipliziert zwei `number_type`-Container. Sie reduziert dazu zunächst die Multiplikation auf das Multiplizieren mit einem einstelligen Faktor. 482*62 wird aufgeteilt in 482*2 und 482*6. Das Produkt der zweiten Ziffer wird in der späteren Summe um eine Stelle verschoben, wodurch dann 482*2+482*60 entsteht.

Die Multiplikation mit einem einstelligen Faktor (beispielsweise 2) verläuft ähnlich. Es werden die Produkte 2*2, 8*2 und 4*2 gebildet. Durch versetzte Addition erhalten wir 2*2+80*2+400*2. Abbildung 42 stellt den Ablauf grafisch dar.

Im Grunde basiert der Algorithmus auf der schriftlichen Multiplikation, wie sie in der Schule zur Genüge durchgeführt werden musste.

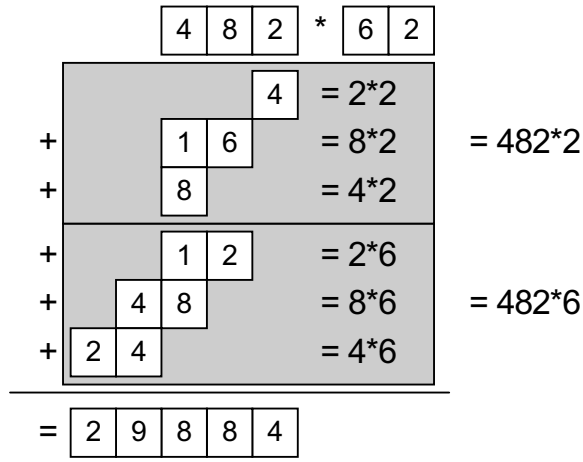


Abbildung 42: Die Funktionsweise von mul

```
CHugeNumber::number_type
CHugeNumber::mul(const CHugeNumber::number_type &v1,
                 const CHugeNumber::number_type &v2) {
  number_type r(1);

/*
** Multiplikation mit einstelligem Faktor?
*/
  if(v2.size()==1) {
    int dvalue=v2.front().m_digit;
    if(dvalue) {
      int addpos=0;

/*
** Bilden der Teilprodukte mit anschließendem
** versetzten Aufaddieren
*/
      for(number_type::const_iterator iter=v1.begin();
          iter!=v1.end();
          ++iter,++addpos)
        r=add(r,CHugeNumber(iter->m_digit*dvalue).m_number,addpos);
      }
  }

/*
** Beliebige Multiplikation
*/
  else {
    int addpos=0;

/*
** Reduktion der Multiplikation auf Multiplikation mit
** einem einstelligen Faktor
** Anschließend werden die Teilprodukte versetzt addiert.
*/
    for(number_type::const_iterator iter=v2.begin();
        iter!=v2.end();
        ++iter,++addpos)
      r=add(r,(v1*iter->m_digit).m_number,addpos);
  }
```

Listing 589: Die private Methode mul

```
/*
** Eventuell führende Nullen werden gekürzt
*/
  shorten(r);
  return(r);
}
```

Listing 589: Die private Methode mul (Forts.)

div

Die Methode `div` implementiert die Division für `CHugeNumber` und nutzt als Algorithmus die schriftliche Division, wie sie in Abbildung 43 dargestellt ist.

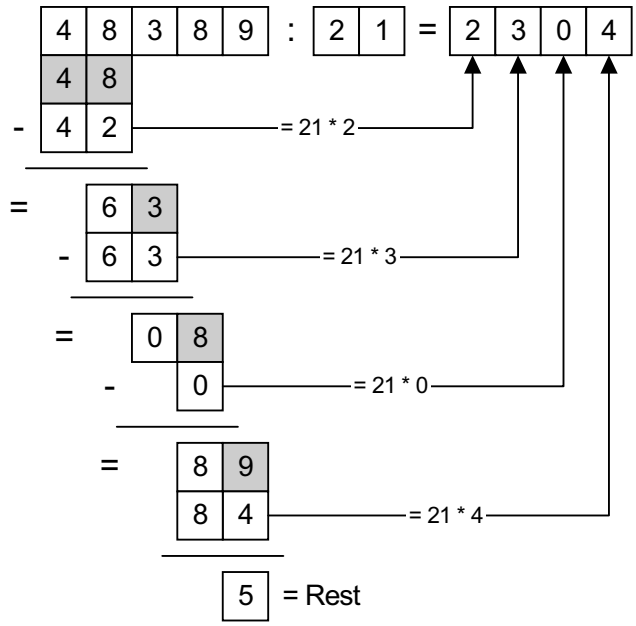


Abbildung 43: Die Funktionsweise von div

```
CHugeNumber::number_type
CHugeNumber::div(const CHugeNumber::number_type &v1,
                 const CHugeNumber::number_type &v2) {

  number_type r;
```

Listing 590: Die private Methode div

```
/*
** v2 größer v1?
** => Ergebnis = 0
*/
  if(compare(v1,v2)>0) {
    r.push_back(0);
    return(r);
  }

  number_type::const_reverse_iterator iter=v1.rbegin();
  number_type part;

  while(true) {

/*
** So lange Ziffern aus der Hauptzahl holen und
** Nullen im Ergebnis eintragen, wie part< v2 gilt
*/
    while((compare(part,v2)>0)&&(iter!=v1.rend())) {

      part.push_front(*(iter++));
      shorten(part);
      if(compare(part,v2)>0)
        r.push_front(0);
    }

/*
** Keine Ziffern mehr in v1 und part noch kleiner v2?
** => Berechnung beendet
*/
    if((iter==v1.rend())&&(compare(part,v2)>0)) {
      shorten(r);
      return(r);
    }

/*
** Bestimmen, wie oft v2 in part passt
*/
    CDigit digit(0);
    while(compare(part,v2)<=0) {
```

Listing 590: Die private Methode div (Forts.)

```
      part=sub(part,v2);
      digit+=1;
    }

/*
** Anzahl als Ziffer zum Ergebnis hinzufügen
*/
    r.push_front(digit);
  }
}
```

Listing 590: Die private Methode div (Forts.)

mod

Wie in Abbildung 43 schön zu sehen ist, ermittelt der Divisions-Algorithmus gleichzeitig – gewissermaßen als Nebenprodukt – auch den Rest der Division.

Die Methode `mod` ist eine abgespeckte Version des Algorithmus, der alle nur für die Bestimmung des Quotienten notwendigen Anweisungen fehlen.

```
CHugeNumber::number_type
CHugeNumber::mod(const CHugeNumber::number_type &v1,
                 const CHugeNumber::number_type &v2) {

/*
** v2 größer v1?
** => Ergebnis = v1
*/
  if(compare(v1,v2)>0)
    return(v1);

  number_type::const_reverse_iterator iter=v1.rbegin();
  number_type part;

  while(true) {

/*
** So lange Ziffern aus der Hauptzahl holen und
** Nullen im Ergebnis eintragen, wie part< v2 gilt
*/
```

Listing 591: Die private Methode mod

```
    while((compare(part,v2)>0)&&(iter!=v1.rend())) {

      part.push_front(*(iter++));
      shorten(part);
    }

/*
** Keine Ziffern mehr in v1 und part noch kleiner v2?
** => Berechnung beendet
*/
    if((iter==v1.rend())&&(compare(part,v2)>0))
      return(part);

/*
** Bestimmen, wie oft v2 in part passt
*/
    while(compare(part,v2)<=0)
      part=sub(part,v2);
  }
}
```

Listing 591: Die private Methode mod (Forts.)

shorten

`shorten` streicht alle führenden Nullen einer Zahl, weil sie bei den meisten Berechnungen und Vergleichen eher stören.

```
void CHugeNumber::shorten(number_type &v) {

/*
** Alle führenden Nullen löschen
** Beim Wert 0 eine Null stehen lassen
*/
  while((v.size()>1)&&(v[v.size()-1]==0))
    v.pop_back();
}
```

Listing 592: Die private Methode shorten

isZero

Diese Methode bietet basierend auf der verwendeten Zahlendarstellung eine schnellere Prüfung auf den Wert 0.

```
bool CHugeNumber::isZero(const CHugeNumber &n) {
  for(CHugeNumber::number_type::const_iterator
                                iter=n.m_number.begin();
      iter!=n.m_number.end();
      ++iter)
    if(iter->m_digit!=0)
      return(false);
  return(true);
}
```

Listing 593: Die private Methode isZero

Rechenoperatoren

Die Rechenoperatoren gestalten sich mit der geleisteten Vorarbeit ziemlich einfach. Lediglich der Additionsoperator ist noch etwas aufwändiger, weil dort die bei `add` fehlende Vorzeichenbetrachtung implementiert werden muss:

```
CHugeNumber operator+(const CHugeNumber &n1,
                      const CHugeNumber &n2) {

/*
** n1 und n2 haben das gleiche Vorzeichen?
** => Vorzeichen übernehmen und Zahlen addieren
*/
  if(n1.m_sign==n2.m_sign)
    return(CHugeNumber(CHugeNumber::add(n1.m_number,n2.m_number),
                       n1.m_sign));

/*
** n1 positiv und n2 negativ?
** => Absolutwerte vergleichen
*/
  if(n1.m_sign==1)

/*
```

Listing 594: operator+ von CHugeNumber

```
** Absolutwert von n1 kleiner als der von n2?
** => Differenz n2-n1 bilden, Vorzeichen ist -
*/
    if(CHugeNumber::compare(n1.m_number,n2.m_number)>0)
      return(CHugeNumber(CHugeNumber::sub(n2.m_number,
                                          n1.m_number),-1));

/*
** Absolutwert von n1 größer als der von n2
** Differenz n1-n2 bilden, Vorzeichen ist +
*/
    else
      return(CHugeNumber(CHugeNumber::sub(n1.m_number,n2.m_number),1));

/*
** n2 positiv und n1 negativ
** Absolutwerte vergleichen
*/
  else

/*
** Absolutwert von n1 kleiner als der von n2?
** => Differenz n2-n1 bilden, Vorzeichen ist +
*/
    if(CHugeNumber::compare(n1.m_number,n2.m_number)>0)
      return(CHugeNumber(CHugeNumber::sub(n2.m_number,n1.m_number),1));

/*
** Absolutwert von n1 größer als der von n2
** Differenz n1-n2 bilden, Vorzeichen ist -
*/
    else
      return(CHugeNumber(CHugeNumber::sub(n1.m_number,
                                          n2.m_number),-1));
}
```

Listing 594: operator+ von CHugeNumber (Forts.)

Die Methode `operator-` macht sich das Leben leicht, indem sie die Subtraktion auf eine Addition zurückführt, bei der das Vorzeichen des zweiten Operanden umgekehrt ist:

```
CHugeNumber operator-(const CHugeNumber &n1,
                      const CHugeNumber &n2) {
  return(n1+CHugeNumber(n2.m_number,-n2.m_sign));
}
```

Listing 595: operator- von CHugeNumber

Die Operatoren für die Multiplikation und Division rufen einfach die entsprechende private Methode auf (`mul` oder `div`) und multiplizieren die Vorzeichen:

```
CHugeNumber operator*(const CHugeNumber &n1,
                      const CHugeNumber &n2) {
  return(CHugeNumber(CHugeNumber::mul(n1.m_number,n2.m_number),
                     n1.m_sign*n2.m_sign));
}

//*********

CHugeNumber operator/(const CHugeNumber &n1,
                      const CHugeNumber &n2) {
  return(CHugeNumber(CHugeNumber::div(n1.m_number,n2.m_number),
                     n1.m_sign*n2.m_sign));
}
```

Listing 596: operator und operator/ von CHugeNumber*

Der Modulo-Operator ließe sich mit der Formel a%b = a-(a/b)*b auf bereits implementierte Operationen zurückführen. Aber die Modulo-Berechnung mit einer Division, Multiplikation und Subtraktion zu erkaufen, wäre zu zeitaufwändig.

Zumal wir bereits eine Methode `mod` geschrieben haben, die diese Aufgabe erledigt. Wir verwenden sie daher auch in `operator%`. Sie ist im Vergleich zu den anderen Rechenoperatoren eine echte Methode:

```
CHugeNumber CHugeNumber::operator%(const CHugeNumber &n) const{
  CHugeNumber result;
  result.m_number=mod(m_number,n.m_number);
  result.m_sign=1;
```

Listing 597: operator% von CHugeNumber

```
  return(result);
}
```

Listing 597: operator% von CHugeNumber (Forts.)

Zuweisungsoperatoren

Die Zuweisungsoperatoren sind von der Funktionalität her identisch mit den Rechenoperatoren, nur dass sich das Ergebnis auf den ersten Operanden auswirkt. Es handelt sich ausnahmslos um Methoden:

```
CHugeNumber &CHugeNumber::operator+=(const CHugeNumber &n) {

/*
** Bei gleichem Vorzeichen das aktuelle Vorzeichen beibehalten
** Und Zahlen addieren
*/
  if(m_sign==n.m_sign) {
    m_number=add(m_number,n.m_number);
    return(*this);
  }

/*
** Wenn eigenes Vorzeichen + und das der anderen Zahl -
** => Ermitteln, welche Zahl den größeren Absolutwert besitzt
*/
  if(m_sign==1)

/*
** Eigener Absolutwert kleiner als der der anderen Zahl?
** => Eigene Zahl von der anderen Zahl abziehen, Vozeichen ist -
*/
    if(CHugeNumber::compare(m_number,n.m_number)>0) {
      m_number=sub(n.m_number,m_number);
      m_sign=-1;
      return(*this);
    }

/*
** Eigener Absolutwert größer als der der anderen Zahl
** Andere Zahl von der eigenen Zahl abziehen, Vorzeichen beibehalten
```

Listing 598: Die Zuweisungsoperatoren von CHugeNumber

```
*/
    else {
      m_number=sub(m_number,n.m_number);
      return(*this);
    }

/*
** Eigenes Vorzeichen ist - und das der anderen Zahl +
** Ermitteln, welche Zahl den größeren Absolutwert besitzt
*/
  else

/*
** Eigener Absolutwert kleiner als der der anderen Zahl?
** => Eigene Zahl von der anderen Zahl abziehen, Vorzeichen ist +
*/
    if(CHugeNumber::compare(m_number,n.m_number)>0) {
      m_number=sub(n.m_number,m_number);
      m_sign=1;
      return(*this);
    }

/*
** Eigener Absolutwert größer als der der anderen Zahl
** Andere Zahl von der eigenen Zahl abziehen, Vorzeichen beibehalten
*/
    else {
      m_number=sub(m_number,n.m_number);
      return(*this);
    }
}

//*********

CHugeNumber &CHugeNumber::operator-=(const CHugeNumber &n) {
  *this+=CHugeNumber(n.m_number,-n.m_sign);
  return(*this);
}

//*********
```

Listing 598: Die Zuweisungsoperatoren von CHugeNumber (Forts.)

```
CHugeNumber &CHugeNumber::operator*=(const CHugeNumber &n) {
  m_number=mul(m_number,n.m_number);
  m_sign*=n.m_sign;
  return(*this);
}

//*********

CHugeNumber &CHugeNumber::operator/=(const CHugeNumber &n) {
  m_number=div(m_number,n.m_number);
  m_sign*=n.m_sign;
  return(*this);
}

//*********

CHugeNumber &CHugeNumber::operator%=(const CHugeNumber &n) {
  m_number=mod(m_number,n.m_number);
//  *this=*this-(*this/n*n);
  m_sign=1;
  return(*this);
}
```

Listing 598: Die Zuweisungsoperatoren von CHugeNumber (Forts.)

compare

Grundlage der Vergleichsoperatoren bildet die Methode `compare`, die zwei `number_type`-Container vergleicht.

Bedingung für einen erfolgreichen Vergleich ist, dass die beiden Zahlen keine führenden Nullen besitzen.

Der Aufruf `compare(n1,n2)` liefert folgenden Wert:

- `1`, wenn n1<n2 gilt
- `0`, wenn n1=n2 gilt
- `-1`, wenn n1>n2 gilt

```
int CHugeNumber::compare(const CHugeNumber::number_type &v1,
                         const CHugeNumber::number_type &v2) {
```

Listing 599: Die private Methode compare

```
/*
** Besitzt eine Zahl mehr Ziffern als die andere,
** muss sie zwangsläufig größer sein.
*/
  if(v1.size()<v2.size())
    return(1);
  if(v1.size()>v2.size())
    return(-1);

/*
** Bei gleicher Ziffernzahl müssen die einzelnen
** Ziffern verglichen werden
*/
  number_type::const_reverse_iterator iter1=v1.rbegin();
  number_type::const_reverse_iterator iter2=v2.rbegin();
  while((iter1!=v1.rend())&&(*iter1==*iter2)) {
    ++iter1;
    ++iter2;
  }

/*
** Unterscheiden sich zwei Ziffern, dann ist die Zahl
** mit der größeren Ziffer auch die größere Zahl.
*/
  if(iter1==v1.rend())
    return(0);
  if((*iter1)<(*iter2))
    return(1);
  else
    return(-1);
}
```

Listing 599: Die private Methode compare (Forts.)

Vergleichsoperatoren

Die Vergleichsoperatoren besitzen die Methode `compare` als Grundlage.

```
bool CHugeNumber::operator<(const CHugeNumber &n) const {
```

Listing 600: Die Vergleichsoperatoren von CHugeNumber

```
/*
** Sind die Vorzeichen der beiden Zahlen unterschiedlich?
** => Zahl mit positivem Vorzeichen ist die größere
*/
  if(m_sign!=n.m_sign)
    if(m_sign==1)
      return(false);
    else
      return(true);
  else

/*
** Sind Vorzeichen gleich, so muss über compare
** entschieden werden.
*/
    if(m_sign==1)
      return(compare(m_number,n.m_number)>0);
    else
      return(compare(m_number,n.m_number)<0);
}

//*********

bool CHugeNumber::operator==(const CHugeNumber &n) const {

/*
** Zwei Zahlen sind gleich, wenn ihr Vorzeichen
** und ihr Absolutwert gleich sind.
*/
  return((m_sign==n.m_sign)&&(compare(m_number,n.m_number)==0));
}

//*********

bool CHugeNumber::operator>(const CHugeNumber &n) const {
  return(n<*this);
}

//*********

bool CHugeNumber::operator!=(const CHugeNumber &n) const {
```

Listing 600: Die Vergleichsoperatoren von CHugeNumber (Forts.)

```
  return(!(*this==n));
}

//*********

bool CHugeNumber::operator>=(const CHugeNumber &n) const {
  return(!(*this<n));
}

//*********

bool CHugeNumber::operator<=(const CHugeNumber &n) const {
  return(!(n<*this));
}
```

Listing 600: Die Vergleichsoperatoren von CHugeNumber (Forts.)

Inkrement- und Dekrementoperatoren

Die Inkrement- und Dekrementoperatoren müssen den Spezialfall beachten, dass bei einem Überschreiten der Null das Vorzeichen wechselt.

```
CHugeNumber &CHugeNumber::operator++() {

/*
** Null?, dann Vorzeichen richtig setzen
*/
  if(isZero(m_number))
    m_sign=1;

/*
** Je nach Vorzeichen erhöhen oder vermindern
*/
  if(m_sign==1)
    increase(m_number);
  else
    decrease(m_number);
  return(*this);
}

//*********
```

Listing 601: Inkrement- und Dekrementoperatoren von CHugeNumber

```
CHugeNumber &CHugeNumber::operator--() {

/*
** Null?, dann Vorzeichen richtig setzen
*/
  if(isZero(m_number))
    m_sign=-1;

/*
** Je nach Vorzeichen erhöhen oder vermindern
*/
  if(m_sign==1)
    decrease(m_number);
  else
    increase(m_number);
  return(*this);
}

//*********

CHugeNumber CHugeNumber::operator++(int) {
  CHugeNumber tmp=*this;
  ++(*this);
  return(tmp);
}

//*********

CHugeNumber CHugeNumber::operator--(int) {
  CHugeNumber tmp=*this;
  --(*this);
  return(tmp);
}
```

Listing 601: Inkrement- und Dekrementoperatoren von CHugeNumber (Forts.)

Verschiebe-Operatoren

Die Verschiebe-Operatoren verschieben ein `CHugeNumber`-Objekt um einzelne Stellen. 334<<3 ergibt 334000 und 12345 >> 2 ergibt 123.

```
CHugeNumber CHugeNumber::operator<<(int w) const{
  CHugeNumber::number_type r(m_number.size()+w);
  CHugeNumber::number_type::iterator riter=r.begin()+w;

/*
** Ziffern um w Stellen nach links schieben und
** freiwerdende Stellen mit 0 füllen
** Vorzeichen wird beibehalten
*/
  for(CHugeNumber::number_type::const_iterator i=m_number.begin();
      i!=m_number.end();
      ++i, ++riter)
    *riter=*i;
  return(r);
}

//*********

CHugeNumber CHugeNumber::operator>>(int w) const{
  CHugeNumber::number_type r(m_number.size()-w);
  CHugeNumber::number_type::iterator riter=r.begin();

/*
** Ziffern um w Stellen nach rechts schieben und
** freiwerdende Stellen mit 0 füllen
** Vorzeichen wird beibehalten
*/
  for(CHugeNumber::number_type::const_iterator i=m_number.begin()+w;
      i!=m_number.end();
      ++i, ++riter)
    *riter=*i;
  return(r);
}
```

Listing 602: Die Verschiebe-Operatoren von CHugeNumber

Ausgabe-Operator

Da die Ausgabe der einzelnen Ziffern in der umgekehrten Speicher-Reihenfolge abläuft, werden Reverse-Iteratoren eingesetzt.

Sollte der Wert der auszugebenden Zahl 0 sein, wird ein eventuell negatives Vorzeichen nicht mit ausgegeben.

```
ostream &operator<<(ostream &ostr, const CHugeNumber &n) {
  if((n.m_sign==-1)&&(!CHugeNumber::isZero(n)))
    ostr << "-";
  for(CHugeNumber::number_type::const_reverse_iterator i=
                                n.m_number.rbegin();
      i!=n.m_number.rend();
      ++i)
    ostr << static_cast<char>(i->m_digit+'0');
  return(ostr);
}
```

getAsString

Zunächst einmal eine ganz einfache Konvertiermethode, die ein `CHugeNumber`-Objekt als String zurückliefert:

```
string CHugeNumber::getAsString() const {
  string str;
  if((m_sign==-1)&&(!CHugeNumber::isZero(m_number)))
    str+="-";
  for(number_type::const_reverse_iterator iter=m_number.rbegin();
      iter!=m_number.rend();
       ++iter)
    str+=iter->m_digit+'0';
  return(str);
}
```

Listing 603: Die Methode getAsString von CHugeNumber

getAsPointedNumber

Damit Ausgaben der Zahl wie zum Beispiel in den Rezepten 95 oder 100 übersichtlicher gestaltet werden, implementieren wir die Methode `getAsPointedNumber`, die beispielsweise die Zahl 34353454 als »34.353.454« liefert.

```
string CHugeNumber::getAsPointedNumber() const {
  string str;
  if((m_sign==-1)&&(!CHugeNumber::isZero(m_number)))
    str+="-";
```

Listing 604: Die Methode getAsPointedNumber

```
  size_t i=m_number.size();
  for(number_type::const_reverse_iterator iter=m_number.rbegin();
      iter!=m_number.rend();
      ++iter, --i) {
    if((i!=m_number.size())&&((i%3)==0))
      str+=".";
    str+=iter->m_digit+'0';
  }
  return(str);
}
```

Listing 604: Die Methode getAsPointedNumber (Forts.)

getAsUnsignedLong

Die Methode wandelt ein `CHugeNumber`-Objekt in einen `unsigned long`-Wert um. Um ein vernünftiges Ergebnis zu erhalten, muss der Benutzer selbst dafür Sorge tragen, dass die im `CHugeNumber`-Objekt gespeicherte Zahl nicht den Wertebereich von `unsigned long` überschreitet.

Ein eventuelles Vorzeichen wird bei der Umwandlung ignoriert.

```
unsigned long CHugeNumber::getAsUnsignedLong() const {
  unsigned long result=0;
  unsigned long bit=1;
  CHugeNumber tmp=*this;
  CHugeNumber two(2);

/*
** Laufe so lange, wie noch Bits in der Zahl gesetzt sind.
*/
  for(int i=0; (i<32)&&(!isZero(tmp)); i++) {

/*
** Ergibt Division durch zwei einen Rest?
** => Das niederwertigste Bit ist gesetzt.
*/
    CHugeNumber div=tmp/two;
    if(div*two<tmp) {
      result|=bit;
    }
```

Listing 605: Die Methode getAsUnsignedLong

```
/*
** Zahl durch zwei teilen und Ergebnis um ein
** Bit nach links schieben
*/
    tmp=div;
    bit<<=1;
  }
  return(result);
}
```

Listing 605: Die Methode getAsUnsignedLong (Forts.)

getAsFileSize

Damit `CHugeNumber` besser mit der Klasse `CFileSize` aus Rezept 84 zusammenarbeitet, fügen wir eine Methode `getAsFileSize` hinzu, die ein `CHugeNumber`-Objekt in ein `CFileSize`-Objekt umwandelt und dieses zurückgibt.

```
CFileSize CHugeNumber::getAsFileSize() const {
  CHugeNumber v("4294967296"), high(*this/v), low(*this%v);
  return(CFileSize(low.getAsUnsignedLong(),
                   high.getAsUnsignedLong()));
}
```

Listing 606: Die Methode getAsFileSize

getAsLong

Der Vollständigkeit halber wird hier noch die Methode `getAsLong` aufgeführt, die ein `CHugeNumber`-Objekt in einen `long`-Wert umwandelt. Auch hier muss der Benutzer selbst sicherstellen, dass die im `CHugeNumber`-Objekt gespeicherte Zahl den Wertebereich eines `long`-Wertes nicht überschreitet.

```
long CHugeNumber::getAsLong() const {
  long result=0;
  long bit=1;
  CHugeNumber tmp=*this;
  CHugeNumber two(2);
```

Listing 607: Die Methode getAsLong

```
/*
** Laufe so lange, wie noch Bits in der Zahl gesetzt sind.
*/
  for(int i=0; (i<32)&&(!isZero(tmp)); i++) {

/*
** Ergibt Division durch zwei einen Rest?
** => Das niederwertigste Bit ist gesetzt.
*/
    CHugeNumber div=tmp/two;
    if(div*two<tmp) {
      result|=bit;
    }

/*
** Zahl durch zwei teilen und Ergebnis um ein
** Bit nach links schieben
*/
    tmp=div;
    bit<<=1;
  }

/*
** Bei negativem Vorzeichen den long-Wert
** mit -1 multiplizieren
*/
  if(m_sign==-1)
    result*=-1;
  return(result);
}
```

Listing 607: Die Methode getAsLong (Forts.)

Die Klasse `CHugeNumber` finden Sie auf der CD in den `CHugeNumber`-Dateien.

Als Beispiel wollen wir die Klasse einmal mit dem Template zur Fakultätsberechnung aus Rezept 117 einsetzen und die Fakultät bis 25 berechnen lassen. Wie Abbildung 44 zeigt, gelangen wir bei der Fakultät von 25 schon zu einem Wert von über 815 Septilliarden.

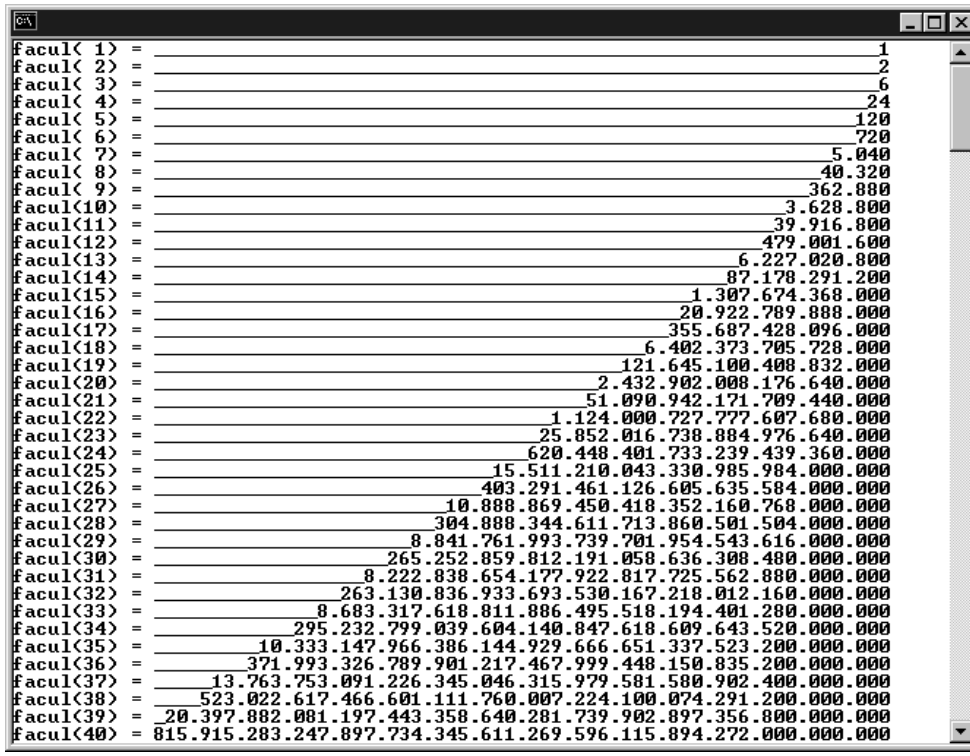


Abbildung 44: Die Fakultäten von 1 bis 40

Das Programm dazu sieht wie folgt aus:

```
#include "CHugeNumber.h"
#include "MathFunctions.h"
#include "StringFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  for(CHugeNumber hncount=1; hncount<=40; hncount++)
    cout << "facul(" << toString(hncount,2) << ") = " <<
         toString(facul(hncount).getAsPointedNumber(),63,false,'_')
         << endl;
}
```

Listing 608: Ein Beispiel zu CHugeNumber

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0708.cpp*.

120 Wie kann eine Zahl in ihre Primfaktoren zerlegt werden?

Wir wollen Zahlen im Wertebereich von `unsigned long` berücksichtigen. Dazu reicht der Algorithmus »Teile und Faktorisiere« aus. Um einen Bereich mit der größten Zahl N mit dem Algorithmus bearbeiten zu können, brauchen wir alle Primzahlen, die kleiner oder gleich der Wurzel von N sind. Zusätzlich benötigen wir mindestens eine Primzahl, die größer oder gleich der Wurzel von N ist.

Bei einem Wertebereich von 2^{32} benötigen wir alle Primzahlen von 2 bis 65535 und die Primzahl 65537, die das zweite Kriterium erfüllt.

An die geforderten Primzahlen könnten wir über die Funktion `getPrimeNumbers` gelangen:

```
vector<unsigned long> primes;
getPrimeNumbers(65537,primes);
```

Will man nur gelegentlich eine Primfaktorzerlegung vornehmen, mag dieser Ansatz ausreichend sein. In dieser Lösung wollen wir die Primzahlen aber fest »verdrahten« und legen ein Feld mit allen 6543 Primzahlen an.

Sie finden dieses Feld auf der CD in der Datei `PrimeNumbers.cpp`.

```
void factorizeIntoPrimes(unsigned long v,
                         vector<unsigned long> &result) {

  if(v<2)
    return;

  unsigned long ppos=0,n=v;

/*
** So lange laufen, wie n ungleich 1
*/
  while(n!=1) {
```

Listing 609: Die Funktion factorizeIntoPrimes

```
/*
** Ergibt die Division durch die aktuelle
** Primzahl einen Rest?
*/
    if(n%primeNumbers[ppos]) {

/*
** Wenn Rest: sollte v geteilt durch die aktuelle Primzahl
** kleiner als die aktuelle Primzahl sein, dann Abbruch
*/
      if((v/primeNumbers[ppos])<primeNumbers[ppos])
        break;

/*
** Andernfalls nächste Primzahl nehmen
*/
      ppos++;
    }
    else {

/*
** Die Division hat keinen Rest ergeben?
** => Es wurde ein Faktor gefunden. Faktor in die Lösung
** übernehmen und mit derselben Primzahl weitermachen
*/
      result.push_back(primeNumbers[ppos]);
      n/=primeNumbers[ppos];
    }
  }

/*
** Sollte der Rest ungleich 1 sein, dann ist er
** der letzte Primfaktor und wird zur Lösung hinzugefügt.
*/
  if(n!=1)
    result.push_back(n);
}
```

Listing 609: Die Funktion factorizeIntoPrimes (Forts.)

Die Funktion `factorize Into Primes` finden Sie auf der CD in den `MathFunctions`-Dateien.

Nachfolgend sehen Sie ein Beispiel zur Primfaktorzerlegung. Abbildung 45 zeigt einen typischen Programmlauf.

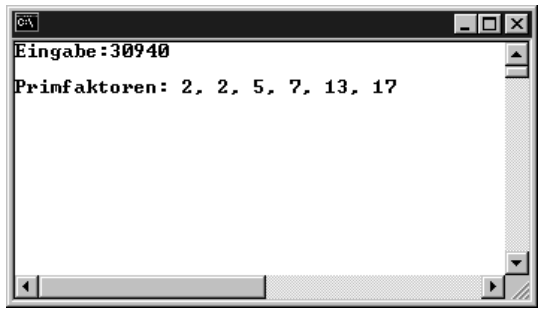


Abbildung 45: Die Primfaktoren von 30940

```
#include "MathFunctions.h"
#include <iostream>

using namespace std;
using namespace Codebook;

int main() {
  cout << "Eingabe:";
  unsigned long number;
  cin >> number;

  cout << "\nPrimfaktoren: ";

  vector<unsigned long> result;
  factorizeIntoPrimes(number,result);

  vector<unsigned long>::iterator iter=result.begin();
  for(cout << *(iter++); iter!=result.end(); ++iter) {
    cout << ", " << *iter;
  }
  cout << endl;
}
```

Listing 610: Ein Beispiel zu factorizeIntoPrimes

Die Schleife zur Ausgabe der Primfaktoren geht davon aus, dass mindestens ein Primfaktor im Vektor gespeichert ist. Auf eine entsprechende Sicherheitsabfrage wurde hier verzichtet.

Sie finden die `main`-Funktion auf der CD unter *Buchdaten\Beispiele\main0709.cpp*.

121 Wie können Matrizen addiert und subtrahiert werden?

Um für die künftigen mathematischen Matrizen-Operationen nicht eine Matrix komplett neu zu entwerfen, verwenden wir die Klasse `CMatrix` aus Rezept 37.

CMathMatrix

Zunächst leiten wir von der STL-Ausnahmen-Klasse `logic_error` unsere eigene Ausnahme `invalid_dimension` ab, um bei späteren Berechnungen ungültige Matrix-Dimensionen mitteilen zu können.

Die Klasse `CMathMatrix` selbst wird von `CMatrix` abgeleitet. Sie bekommt keine zusätzlichen Attribute, weil die Klasse `CMatrix` die gesamte Matrizen-Funktionalität abdeckt. `CMathMatrix` wird lediglich Methoden zur Berechnung mathematischer Matrizen-Operationen beinhalten.

Klassendefinition

```
class invalid_dimension : public std::logic_error {
public:
  explicit invalid_dimension(const std::string& error)
  : logic_error(error)
  { }

};

template<typename Type, typename Allocator=std::allocator<Type> >
class CMathMatrix : public CMatrix<Type, Allocator> {
public:
  typedef CMatrix<Type, Allocator> base_type;
  typedef CMathMatrix<Type, Allocator> my_type;
  typedef base_type::size_type size_type;
  typedef base_type::reference reference;
```

Listing 611: Die Klassendefinition von CMathMatrix

```
  typedef base_type::const_reference const_reference;
};
```

Listing 611: Die Klassendefinition von CMathMatrix (Forts.)

Konstruktoren

Mit den Konstruktoren sollen die gleichen Möglichkeiten geboten werden, die `CMatrix` bereits unterstützt. Wir schleifen die Funktionalität dazu einfach durch.

```
CMathMatrix()
: base_type()
{}

//*********

CMathMatrix(const CMathMatrix &m)
: base_type(m)
{}

//*********

CMathMatrix(size_type rows,
            size_type columns,
            const Type &obj=Type())
: base_type(rows, columns, obj)
{}

//*********

CMathMatrix(const CMathMatrix &m,
            size_type rbegin, size_type cbegin,
            size_type rows, size_type columns)
: base_type(m,rbegin,cbegin,rows,columns)
{}

//*********

template<typename IType>
CMathMatrix(bool column, IType beg, IType end)
```

Listing 612: Die Konstruktoren von CMathMatrix

```
: base_type(column, beg, end)
{}
```

Listing 612: Die Konstruktoren von CMathMatrix (Forts.)

add

Zwei Matrizen werden addiert, indem die einzelnen Elemente addiert werden. Dazu müssen die beiden Matrizen vom selben Typ sein. Die Methode bekommt zwei Matrizen übergeben, wobei die zweite Matrix zur ersten hinzuaddiert wird.

Die Methoden von `CMathMatrix` verwenden durchgängig den Aufruf-Operator für den Elementzugriff. Dadurch wird die Implementierung nicht langsamer, bei einer anderen Basis-Klasse aber unter Umständen schneller.

```
void add(CMathMatrix &res, const CMathMatrix &op) const {

/*
** Beide Matrizen haben nicht denselben Typ?
** => Ausnahme werfen
*/
  if((res.getRows()!=op.getRows())||
     (res.getColumns()!=op.getColumns()))
    throw invalid_dimension("CMathMatrix::add");

/*
** Die einzelnen Elemente addieren
*/
  for(size_type r=0; r<res.getRows(); ++r)
    for(size_type c=0; c<res.getColumns(); ++c)
      res(r,c)+=op(r,c);
}
```

Listing 613: Die private Methode add von CMathMatrix

sub

Die Methode `sub` bekommt zwei Matrizen übergeben und subtrahiert die zweite von der ersten.

```
void sub(CMathMatrix &res, const CMathMatrix &op) const {

/*
** Beide Matrizen haben nicht denselben Typ?
** => Ausnahme werfen
*/
  if((res.getRows()!=op.getRows())||
     (res.getColumns()!=op.getColumns()))
    throw invalid_dimension("CMathMatrix::sub");

/*
** Die einzelnen Elemente subtrahieren
*/
  for(size_type r=0; r<res.getRows(); ++r)
    for(size_type c=0; c<res.getColumns(); ++c)
      res(r,c)-=op(r,c);
}
```

Listing 614: Die private Methode sub von CMathMatrix

Rechenoperatoren

Die Rechenoperatoren selbst sind trivial, weil sie auf die Methoden `add` und `sub` zurückgreifen, die die eigentliche Arbeit übernehmen.

```
CMathMatrix operator+(const CMathMatrix &m) const {
  CMathMatrix res(*this);
  add(res,m);
  return(res);
}

//*********

CMathMatrix &operator+=(const CMathMatrix &m) {
  add(*this,m);
  return(*this);
}

//*********

CMathMatrix operator-(const CMathMatrix &m) const {
```

Listing 615: Die Additions- und Subtraktionsoperatoren von CMathMatrix

```
  CMathMatrix res(*this);
  sub(res,m);
  return(res);
}

//*********

CMathMatrix &operator-=(const CMathMatrix &m) {
  sub(*this,m);
  return(*this);
}
```

Listing 615: Die Additions- und Subtraktionsoperatoren von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

122 Wie können Matrizen verglichen werden?

Zwei Matrizen sind genau dann gleich, wenn sie vom selben Typ sind und alle ihre Elemente übereinstimmen.

Wir ergänzen dazu die Klasse `CMathMatrix` um die Operatoren `==` und `!=`:

```
bool operator==(const CMathMatrix &m) {
  if((getColumns()!=m.getColumns())||
     (getRows()!=m.getRows()))
    return(false);
  for(size_type r=0; r<getRows(); ++r)
    for(size_type c=0; c<getColumns(); ++c)
      if((*this)(r,c)!=m(r,c))
        return(false);
  return(true);
}

//*********

bool operator!=(const CMathMatrix &m) {
  return(!((*this)==m));
}
```

Listing 616: Die Vergleichsoperatoren von CMathMatrix

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

123 Wie können Matrizen multipliziert werden?

Die Matrizen-Multiplikation ist nicht kommutativ (a*b ist damit nicht b*a.) Zwei Matrizen können nur dann miteinander multipliziert werden, wenn die Zeilenanzahl der ersten Matrix mit der Spaltenanzahl der zweiten Matrix übereinstimmt (die beiden Typen (n,m) und (m,o) sind)

Das Ergebnis ist dann vom Typ (n,o).

Die Elemente der Ergebnis-Matrix werden berechnet, indem für jedes Element e(i,k) das skalare Produkt der i-ten Zeile der ersten Matrix mit der k-ten Spalte der zweiten Matrix bestimmt wird.

Wir fügen eine Methode `mul` zur Matrix-Klasse `CMathMatrix` aus Rezept 121 hinzu:

mul

Die Methode hat drei Parameter: die Ergebnis-Matrix und die beiden Faktoren.

```
void mul(CMathMatrix &res,
         const CMathMatrix &op1,
         const CMathMatrix &op2) const {

/*
** Es kann nur eine Matrix vom Typ (n,m) mit einer Matrix
** vom Typ (m,o) multipliziert werden
*/
  if(op1.getColumns()!=op2.getRows())
    throw invalid_dimension("CMathMatrix::mul");

/*
** Die Ergebnis-Matrix ist vom Typ (n,o)
*/
  res.resize(op1.getRows(), op2.getColumns());

/*
** Jedes Element e(i,k) von res ist skalares Produkt
** der i-ten Zeile von op1 mit der k-ten Spalte von op2
*/
  for(size_type r=0;r<res.getRows();++r)
```

Listing 617: Die Methode mul für zwei Matrizen

```
    for(size_type c=0;c<res.getColumns();++c) {
      Type newElement=0;

/*
** Skalares Produkt bilden
*/
      for(size_type v=0;v<op1.getColumns();++v)
        newElement+=(op1(r,v)*op2(v,c));
      res(r,c)=newElement;
    }
}
```

Listing 617: Die Methode mul für zwei Matrizen (Forts.)

Um eine Matrix mit einem skalaren Wert zu multiplizieren, muss jedes Element der Matrix mit dem skalaren Wert multipliziert werden:

Wir überladen dazu die Methode `mul` mit einer Variante, die die Matrix und den skalaren Wert übergeben bekommt.

```
void mul(CMathMatrix &res, const Type &op) const {

/*
** Jedes Element der Matrix wird mit op multipliziert
*/
  for(size_type r=0;r<res.getRows();++r)
    for(size_type c=0;c<res.getColumns();++c)
      res(r,c)*=op;
}
```

Listing 618: Die Methode mul für eine Matrix und ein skalares Produkt

Rechenoperatoren

Um die Multiplikation für den Benutzer der Klasse zur Verfügung zu stellen, überladen wir die Multiplikationsoperatoren:

```
CMathMatrix operator*(const CMathMatrix &m) const {
  CMathMatrix res;
  mul(res,*this,m);
```

Listing 619: Die Multiplikationsoperatoren von CMathMatrix

```
  return(res);
}

//*********

CMathMatrix &operator*=(const CMathMatrix &m) {
  CMathMatrix res;
  mul(res,*this,m);
  *this=res;
  return(*this);
}

//*********

CMathMatrix operator*(const Type &obj) const {
  CMathMatrix res(*this);
  mul(res,obj);
  return(res);
}

//*********

CMathMatrix &operator*=(const Type &obj) {
  mul(*this,obj);
  return(*this);
}
```

Listing 619: Die Multiplikationsoperatoren von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

124 Wie wird geprüft, ob eine Matrix quadratisch ist?

Diese Aufgabe ist leicht zu bewältigen. Eine Matrix ist genau dann quadratisch, wenn sie genau so viele Zeilen wie Spalten hat.

Die Matrix-Klasse `CMathMatrix` aus Rezept 121 bekommt eine Methode `isQuadratic`, die diese Aufgabe erfüllt:

isQuadratic

```
bool isQuadratic() const {
  return(getRows()==getColumns());
}
```

Listing 620: Die Methode isQuadratic von CMathMatrix

Die Klasse `CMathMatrix` finden Sie auf der CD in der `CMathMatrix.h`-Datei.

125 Wie wird geprüft, ob eine Matrix eine Einheitsmatrix ist?

Als Einheitsmatrizen kommen nur quadratische Matrizen in Frage. Bei ihnen müssen alle Elemente auf der Hauptdiagonalen[1] den Wert 1 und die übrigen Elemente den Wert 0 aufweisen.

isUnitMatrix

Die Methode `isUnitMatrix` von `CMathMatrix` aus Rezept 121 nimmt diese Prüfung vor:

```
bool isUnitMatrix() const {

/*
** Nur quadratische Matrizen können Einheitsmatrizen sein.
*/
  if(!isQuadratic())
    return(false);

/*
** Die Elemente auf der Hauptdiagonalen müssen den Wert 1,
** alle anderen den Wert 0 haben.
*/
  for(size_type r=0;r<getRows();++r)
    for(size_type c=0;c<getColumns();++c)
      if(((r==c)&&((*this)(r,c)!=1))||
```

Listing 621: Die Methode isUnitMatrix von CMathMatrix

1. Die Hauptdiagonale bilden alle Elemente e(i,i) einer quadratischen Matrix.

```
          ((r!=c)&&((*this)(r,c)!=0)))
          return(false);
  return(true);
}
```

Listing 621: Die Methode isUnitMatrix von CMathMatrix (Forts.)

Häufig kommt es vor, dass durch kleine Ungenauigkeiten bei der Berechnung (speziell bei Fließkommazahlen) die Werte nicht exakt 0 oder 1 sind. Wir überladen dazu die Methode `isUnitMatrix` mit einer Variante, der ein Epsilon-Wert übergeben werden kann.

Sollten die Werte in der Matrix nicht mehr als Epsilon von den Werten 0 und 1 abweichen, dann gilt die Matrix als Einheitsmatrix:

```
bool isUnitMatrix(const Type &eps) const {

/*
** Nur quadratische Matrizen können Einheitsmatrizen sein.
*/
  if(!isQuadratic())
    return(false);

/*
** Die Elemente auf der Hauptdiagonalen müssen den Wert 1+/-eps,
** alle anderen den Wert 0+/-eps haben.
*/
  for(size_type r=0;r<getRows();++r)
    for(size_type c=0;c<getColumns();++c)
      if(((r==c)&&(fabs((*this)(r,c)-1)>=eps))||
          ((r!=c)&&(fabs((*this)(r,c))>=eps)))
          return(false);
  return(true);
}
```

Listing 622: Die Methode isUnitMatrix mit Epsilon-Wert

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

126 Wie wird eine Matrix in eine Einheitsmatrix umgewandelt?

Sollte die umzuwandelnde Matrix eine quadratische Matrix sein, dann werden alle Elemente der Hauptdiagonalen auf den Wert 1 und die anderen Elemente auf den Wert 0 gesetzt.

Wir ergänzen die Klasse `CMathMatrix` aus Rezept 121 um die Methode `convertToUnitMatrix`:

```
void convertToUnitMatrix() {

/*
** Nur eine quadratische Matrix kann in
** eine Einheitsmatrix umgewandelt werden.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::convertToE");

/*
** Die Elemente auf der Hauptdiagonalen werden auf 1,
** alle anderen auf 0 gesetzt.
*/
  for(size_type r=0;r<getRows();++r)
    for(size_type c=0;c<getColumns();++c)
      if(r==c)
        (*this)(r,c)=1;
      else
        (*this)(r,c)=0;
}
```

Listing 623: Die Methode convertToUnitMatrix von CMathMatrix

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

127 Wie wird eine Matrix transponiert?

Üblicherweise können nur quadratische Matrizen transponiert werden. Dabei werden die Werte der Elemente e(i,k) und e(k,i) vertauscht.

Wir werden aber auch das Transponieren von nicht quadratischen Matrizen zulassen. Auf diese Weise ändert sich der Typ einer Matrix (n,m) in den Typ (m,n).

Damit lassen sich dann bequem Spaltenvektoren in Zeilenvektoren umwandeln und umgekehrt.

Die Klasse `CMathMatrix` aus Rezept 121 wird dazu mit der Methode `transpose` ausgestattet.

transpose

```
void transpose() {
  if(isQuadratic()) {

/*
** Ist Matrix quadratisch?
** => Die elemente e(r,c) und e(c,r) werden vertauscht.
*/
    for(size_type r=0;r<getRows();r++)
      for(size_type c=r+1;c<getColumns();c++)
        swap((*this)(c,r),(*this)(r,c));
  }
  else {

/*
** Ist Matrix nicht quadratisch?
** => Eine neue Matrix für das Ergebnis wird angelegt.
*/
    CMathMatrix res(getColumns(), getRows());

/*
** Das Element e(c,r) der transponierten Matrix bekommt
** den Wert von Element e(r,c) der originalen Matrix.
*/
    for(size_type r=0;r<getRows();r++)
      for(size_type c=0;c<getColumns();c++)
        res(c,r)=(*this)(r,c);
    *this=res;
  }
}
```

Listing 624: Die Methode transpose von CMathMatrix

getTransposedMatrix

Um eine transponierte Matrix zu berechnen, ohne die Originalmatrix zu zerstören, fügen wir noch die Methode `getTransposedMatrix` hinzu:

```
CMathMatrix getTransposedMatrix() const {
  CMathMatrix res(getColumns(), getRows());

/*
** Das Element e(c,r) der transponierten Matrix bekommt
** den Wert von Element e(r,c) der originalen Matrix.
*/
  for(size_type r=0;r<getRows();r++)
    for(size_type c=0;c<getColumns();c++)
      res(c,r)=(*this)(r,c);
  return(res);
}
```

Listing 625: Die Methode getTransposedMatrix von CMathMatrix

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

128 Wie wird eine Unterdeterminante ermittelt?

Betrachten wir eine Matrix als Determinante, dann entsteht die Unterdeterminante U(i,k) durch das Entfernen der i-ten Zeile und k-ten Spalte der ursprünglichen Determinante.

Ist beispielsweise eine dreireihige Determinante gegeben:

$$D = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

Dann lassen sich neun Unterdeterminanten bilden, wobei wir eine genauer anschauen wollen:

$$U_{21} = \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix}$$

getSubdeterminant

Die Methode `getSubdeterminant` erledigt dies für uns:

```
CMathMatrix getSubdeterminant(size_type drow,
                              size_type dcolumn) const {
```

Listing 626: Die Methode getSubdeterminant von CMathMatrix

```
/*
** Determinanten müssen quadratisch sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::getSubdeterminant");

/*
** Die Subdeterminante hat genau eine Zeile und Spalte
** weniger als das Original
*/
  CMathMatrix res(getRows()-1, getColumns()-1);
  for(int dr=0, sr=0; dr<res.getRows(); dr++, sr++) {

/*
** Zu streichende Zeile erreicht?
** => überspringen
*/
    if(sr==drow)
      sr++;
    for(int dc=0,sc=0;dc<res.getColumns(); dc++, sc++) {

/*
** Zu streichende Spalte erreicht?
** => überspringen
*/
      if(sc==dcolumn)
        sc++;
      res(dr,dc)=(*this)(sr,sc);
    }
  }
  return(res);
}
```

Listing 626: Die Methode getSubdeterminant von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

129 Wie wird eine Determinante berechnet?

Eine zweireihige Determinante ist leicht berechnet. Man subtrahiert das Produkt der Nebendiagonalen von dem Produkt der Hauptdiagonalen.

$$D_2 = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

Bei den Formeln ist zu berücksichtigen, dass die Indizes bei der Determinanten-Darstellung in der Mathematik mit 1 beginnen. Deswegen ist ein entsprechender Transfer auf die mit 0 beginnenden Indizes der C++-Matrix notwendig.

Etwas aufwändiger gestaltet sich die Berechnung einer n-reihigen Determinanten. Dazu wählt man eine beliebige Reihe der Determinanten aus und addiert die Produkte von Elementen und dazugehöriger Adjunkte.

$D_n = \sum_{x=1}^{n} a_{ix} A_{ix}$, mit n in diesem Fall einer beliebigen Zeile

Wie die Adjunkte berechnet wird, steht in Rezept 130.

calcDeterminant

Die Methode `calcDeterminant` setzt die oberen Formeln um:

```
Type calcDeterminant(void) const {

/*
** Determinanten müssen quadratisch sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::calcDeterminant");

/*
** Determinante zweireihig?
** => Wert = e(0,0)*e(1,1) - e(1,0)*e(0,1)
*/
  if(getRows()==2)
    return(((*this)(0,0) * (*this)(1,1))-
           ((*this)(0,1) * (*this)(1,0)));

/*
** Determinante mehr als zweireihig?
** => Werte der ersten Spalte mit ihrer Adjunkten
```

Listing 627: Die Methode calcDeterminant von CMathMatrix

```
** multiplizieren und aufaddieren
*/
  else {
    Type res=0;
    for(size_type r=0;r<getRows(); r++)
      res+=(*this)(r,0)*calcAdjoint(r,0);

    return(res);
  }
}
```

Listing 627: Die Methode calcDeterminant von CMathMatrix (Forts.)

Die Methode macht von `calcAdjoint` aus Rezept 130 Gebrauch.

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

130 Wie wird eine Adjunkte berechnet?

Die Adjunkte A_{ik} einer Determinanten kann mit folgender Formel bestimmt werden:

$A_{ik} = U_{ik}(-1)^{i+k}$, wobei U_{ik} die entsprechende Unterdeterminante ist.

Mit den Methoden `getSubdeterminant` aus Rezept 128 und `calcDeterminant` aus Rezept 129 lässt sich eine Unterdeterminante leicht berechnen:

```
getSubdeterminant(i-1, k-1).calcDeterminant()
```

Die Indizes müssen um 1 vermindert werden, weil die Indizes von `CMathMatrix` bei 0 beginnen.

calcAdjoint

```
Type calcAdjoint(size_type drow, size_type dcolumn) const {

/*
** Determinanten müssen quadratisch sein.
*/
```

Listing 628: Die Methode calcAdjoint von CMathMatrix

```
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::calcAdjoint");

/*
** Vorzeichen der Adjunkten berücksichtigen
*/
  if((drow+dcolumn)&1)
    return(-getSubdeterminant(drow, dcolumn).calcDeterminant());
  else
    return(getSubdeterminant(drow, dcolumn).calcDeterminant());
}
```

Listing 628: Die Methode calcAdjoint von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

131 Wie kann eine Matrix auf Regularität geprüft werden?

Eine Matrix ist genau dann regulär, wenn ihre Determinante einen von Null verschiedenen Wert hat. Wegen Ungenauigkeiten in der Berechnung werden wir die Methode `isRegular` zweimal implementieren, einmal mit Epsilon-Wert.

isRegular

```
bool isRegular() const {

/*
** Nur eine quadratische Matrix kann regulär sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::isRegular");

  return(calcDeterminant()!=0);
}

//*********

bool isRegular(const Type &eps) const {
```

Listing 629: Die Methode isRegular von CMathMatrix

```
/*
** Nur eine quadratische Matrix kann regulär sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::isRegular");

  return(fabs(calcDeterminant())>eps);
}
```

Listing 629: Die Methode isRegular von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

132 Wie kann eine Matrix auf Singularität geprüft werden?

Die Singularität ist gewisserweise das Gegenstück zur Regularität. Eine Matrix ist genau dann singulär, wenn ihre Determinante den Wert 0 hat.

Auch hier werden wir von `isSingular` eine zusätzliche Variante mit Epsilon-Wert implementieren.

isSingular

```
bool isSingular() const {

/*
** Nur eine quadratische Matrix kann singulär sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::isSingular");

  return(calcDeterminant()==0);
}

//*********

bool isSingular(const Type &eps) const {
```

Listing 630: Die Methode isSingular von CMathMatrix

```
/*
** Nur eine quadratische Matrix kann singulär sein.
*/
  if(!isQuadratic())
    throw invalid_dimension("CMathMatrix::isSingular");

  return(fabs(calcDeterminant())<=eps);
}
```

Listing 630: Die Methode isSingular von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

133 Wie wird von einer Matrix die inverse Matrix berechnet?

Eine Matrix A^{-1} ist genau dann die inverse Matrix einer Matrix A, wenn gilt:

$A^{-1}A = AA^{-1} = E$, wobei E die Einheitsmatrix ist.

Mit der Einschränkung, dass eine inverse Matrix nur von einer regulären quadratischen Matrix berechnet werden kann, wird folgende Formel verwendet.

$$A^{-1} = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{21} & \vdots & A_{n1} \\ A_{12} & A_{22} & \vdots & A_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ A_{1n} & A_{2n} & \vdots & A_{nn} \end{pmatrix}$$

Dabei ist A^{ik} die entsprechende Adjunkte. Es ist zu beachten, dass in der oberen Formel das Element a^{ik} durch die Adjunkte A^{ki} ersetzt wurde. Die Adjunkte hat also in Bezug auf das Element vertauschte Indizes.

getInverseMatrix

```
CMathMatrix getInverseMatrix() const {

/*
** Inverse Matrizen nur bei regulären quadratischen Matrizen bestimmbar
*/
  if((!isQuadratic())||(!isRegular()))
    throw invalid_dimension("CMathMatrix::getInverseMatrix");
```

Listing 631: Die Methode getInverseMatrix von CMathMatrix

```
/*
** Matrix zur Aufnahme der Berechnungen anlegen
*/
  CMathMatrix res(getRows(), getColumns());

/*
** Jedes Element wird durch die entsprechende Adjunkte ersetzt.
** (Vertauschte Indizes!)
*/
  for(size_type r=0; r<getRows(); r++)
    for(size_type c=0; c<getColumns(); c++)
      res(r,c)=calcAdjoint(c,r);

/*
** Die Matrix wird mit 1/det multipliziert
*/
  res*=(1/calcDeterminant());
  return (res);
}
};
```

Listing 631: Die Methode getInverseMatrix von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

134 Wie kann eine Matrix auf Symmetrie geprüft werden?

Eine quadratische Matrix ist symmetrisch, wenn jeweils die Elemente e(i,k) und e(k,i) denselben Wert besitzen. Wir werden die Methode isSymmetric einmal mit und einmal ohne Epsilon-Wert implementieren.

isSymmetric

```
bool isSymmetric() const {

/*
** Nur eine quadratische Matrix kann symmetrisch sein.
```

Listing 632: Die Methode isSymmetric von CMathMatrix

```
*/
  if(!isQuadratic())
    return(false);

  for(size_type r=0;r<getRows();r++)
    for(size_type c=r+1;c<getColumns();c++)

/*
** e(r,c)==e(c,r)?
*/
      if((*this)(r,c)!=(*this)(c,r))
        return(false);

  return(true);
}

//*********

bool isSymmetric(const Type &eps) const {

/*
** Nur eine quadratische Matrix kann symmetrisch sein.
*/
  if(!isQuadratic())
    return(false);

  for(size_type r=0;r<getRows();r++)
    for(size_type c=r+1;c<getColumns();c++)

/*
** Unterscheiden sich e(r,c) und e(c,r) um mehr als eps?
*/
      if(fabs((*this)(r,c)-(*this)(c,r))>eps)
        return(false);

  return(true);
}
```

Listing 632: Die Methode isSymmetric von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

135 Wie kann eine Matrix auf Schiefsymmetrie geprüft werden?

Eine Matrix ist genau dann schiefsymmetrisch, wenn die Elemente der Hauptdiagonalen alle den Wert 0 besitzen und sich die Elemente e(i,k) und e(k,i) mit i ungleich k nur im Vorzeichen unterscheiden.

isSkewSymmetric

Wir implementieren die Methode `isSkewSymmetric` einmal mit und einmal ohne Epsilon-Wert.

```
bool isSkewSymmetric() const {

/*
** Nur eine quadratische Matrix kann schiefsymmetrisch sein.
*/
  if(!isQuadratic())
    return(false);

  for(size_type r=0;r<getRows();r++)
    for(size_type c=r;c<getColumns();c++)

/*
** Ist Hauptdiagonal-Element ungleich 0?
*/
      if(((c==r)&&((*this)(r,c)!=0))||((c!=r)&&

/*
** Unterscheiden sich e(r,c) und e(c,r) auch im Absolutwert?
*/
          (-(*this)(r,c)!=(*this)(c,r) )))
        return(false);

  return(true);
}

//*********

bool isSkewSymmetric(const Type &eps) const {
```

Listing 633: Die Methode isSkewSymmetric von CMathMatrix

```
/*
** Nur eine quadratische Matrix kann schiefsymmetrisch sein.
*/
  if(!isQuadratic())
    return(false);

  for(size_type r=0;r<getRows();r++)
    for(size_type c=r;c<getColumns();c++)

/*
** Ist Hauptdiagonal-Element größer eps?
*/
      if(((c==r)&&(fabs((*this)(r,c))>eps))||((c!=r)&&

/*
** Unterscheiden sich die Absolutwerte von e(r,c)
** und e(c,r) um mehr als eps?
*/
          (fabs((*this)(r,c)+(*this)(c,r))>eps )))
        return(false);

  return(true);
}
```

Listing 633: Die Methode isSkewSymmetric von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

136 Wie kann geprüft werden, ob eine Matrix eine obere Dreiecksmatrix ist?

Eine quadratische Matrix ist genau dann eine obere Dreiecksmatrix, wenn alle Elemente unter der Hauptdiagonalen den Wert 0 besitzen.

isUpperTriangularMatrix

Wie bisher bei allen prüfenden Methoden wollen wir wieder zwei Varianten implementieren. Einmal mit und einmal ohne Epsilon-Wert.

```
bool isUpperTriangularMatrix() const {

/*
** Nur eine quadratische Matrix kann eine obere Dreiecksmatrix sein.
*/
  if(!isQuadratic())
    return(false);

  for(size_type c=0;c<getColumns();c++)
    for(size_type r=c+1;r<getRows();r++)
      if((*this)(r,c)!=0)
        return(false);

  return(true);
}

//*********

bool isUpperTriangularMatrix(const Type &eps) const {

/*
** Nur eine quadratische Matrix kann eine obere Dreiecksmatrix sein.
*/
  if(!isQuadratic())
    return(false);

  for(size_type c=0;c<getColumns();c++)
    for(size_type r=c+1;r<getRows();r++)
      if(fabs((*this)(r,c))>eps)
        return(false);

  return(true);
}
```

Listing 634: Die Methode isUpperTriangularMatrix von CMathMatrix

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

137 Wie kann ein lineares Gleichungssystem gelöst werden?

Sollte ein lineares Gleichungssystem der folgenden Form gegeben sein:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
\vdots \quad + \quad \vdots \quad + \quad \vdots \quad + \quad \vdots \quad &= \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}$$

Dann lässt sich dieses Gleichungssystem in eine Matrizenschreibweise übertragen:

$$\left(\begin{array}{ccc} a_{11} & a_{12} & a_{1n} \\ a_{21} & a_{22} & a_{2n} \\ \hline a_{n1} & a_{n2} & a_{nn} \end{array}\right)\left(\begin{array}{c} x_1 \\ x_2 \\ \hline x_n \end{array}\right) = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline b_n \end{array}\right)$$

Der erste Schritt besteht nun darin, die Koeffizientenmatrix in eine obere Dreiecksmatrix zu übertragen. Das Gleichungssystem hat dann die Form

$$\left(\begin{array}{ccc} a_{11} & a_{12} & a_{1n} \\ 0 & a_{22} & a_{2n} \\ \hline 0 & 0 & a_{nn} \end{array}\right)\left(\begin{array}{c} x_1 \\ x_2 \\ \hline x_n \end{array}\right) = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline b_n \end{array}\right)$$

was dem folgenden Gleichungssystem entspricht:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
\vdots \quad &= \vdots \\
a_{nn}x_n &= b_n
\end{aligned}$$

gaussElimination

Das hier zum Zuge kommende Verfahren ist das Gauss'sche Eliminationsverfahren. Wir implementieren dazu eine Methode `gaussElimination`, die die Koeffizientenmatrix und den Ergebnisvektor als `pair` übergeben bekommt und die umgewandelten Objekte wieder als `pair` zurückliefert.

```
static std::pair<CMathMatrix,CMathMatrix>
gaussElimination(const std::pair<CMathMatrix,CMathMatrix> &p) {
```

Listing 635: Die Methode gaussElimination von CMathMatrix

```
/*
** Koeffizientenmatrix muss quadratisch sein und die gleiche
** Zeilenanzahl haben wie der Ergebnisvektor
*/
  if(!p.first.isQuadratic()||(p.first.getRows()!=p.second.getRows()))
    throw invalid_dimension("CMathMatrix::gaussElimination");

  CMathMatrix k=p.first, e=p.second;

/*
** Das betragsgrößte Element der zu elimierenden Spalte
** suchen
*/
  for(size_type step=0; step<k.getRows()-1; step++) {
    size_type posofmaxval=step;
    for(size_type r=step+1; r<k.getRows(); r++)
      if(fabs(k(posofmaxval,step))<fabs(k(r,step)))
        posofmaxval=r;

/*
** Die aktuelle Zeile und die Zeile mit dem
** betragsgrößten Element vertauschen
*/
    if(posofmaxval!=step) {
      k.swapRows(step,posofmaxval);
      e.swapRows(step,posofmaxval);
    }

/*
** Die aktuelle Spalte wird eliminiert.
*/
    for(size_type transgl=step+1;transgl<k.getRows();transgl++) {
      Type factor=-k(transgl,step)/k(step,step);

      e(transgl,0)=e(transgl,0)+(e(step,0)*factor);
      for(size_type c=step; c<k.getRows(); c++) {
        k(transgl,c)=k(transgl,c)+(k(step,c)*factor);
      }
    }
  }
```

Listing 635: Die Methode gaussElimination von CMathMatrix (Forts.)

```
/*
** Umgewandelte Koeffizientenmatrix und angepassten
** Ergebnisvektor zurückliefern
*/
  return(make_pair(k,e));
}
```

Listing 635: Die Methode gaussElimination von CMathMatrix (Forts.)

swapRows

`gaussElimination` macht von der Methode `swapRows` Gebrauch, die zwei Zeilen der Matrix vertauscht:

```
void swapRows(size_type r1, size_type r2) {
  swap((*this)[r1], (*this)[r2]);
}
```

Listing 636: Die Methode swapRows von CMathMatrix

Der nächste Schritt besteht nun darin, die Koeffizientenmatrix und den Ergebnisvektor in folgende Form zu überführen:

$$\left(\begin{array}{cc:c} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \hdashline 0 & 0 & 1 \end{array}\right)\left(\begin{array}{c} x_1 \\ x \\ \hdashline x_n \end{array}\right)=\left(\begin{array}{c} b_1 \\ b_2 \\ \hdashline b_n \end{array}\right)$$

Das entspricht dem Gleichungssystem

$$\begin{aligned} x_1 &= b_1 \\ x_2 &= b_2 \\ \vdots &= \vdots \\ x_n &= b_n \end{aligned}$$

Der ursprüngliche Ergebnisvektor ist damit in den Lösungsvektor umgewandelt worden.

calcSolutionVector

```
static CMathMatrix
calcSolutionVector(const std::pair<CMathMatrix,CMathMatrix> &p,
                   const Type &eps=0) {

/*
** Koeffizientenmatrix muss eine obere Dreiecksmatrix sein
** Zeilenanzahl von Koeffizientenmatrix und Ergebnisvektor
** müssen übereinstimmen.
*/
  if((!p.first.isUpperTriangularMatrix(eps))||
     (p.first.getRows()!=p.second.getRows()))
    throw invalid_dimension("CMathMatrix::calcSolutionVector");

  CMathMatrix k=p.first, e=p.second;
  size_type n=k.getRows();

  cout << "n:" << n << endl;
  size_type r=n-1;

/*
** Das Gleichungssystem wird zeilenweise aufgelöst.
*/
  while(true) {
    for(size_type c=n-1;c>r;c--) {
      k(r,c)=k(r,c)*e(c,0);
      e(r,0)=e(r,0)-k(r,c);
      k(r,c)=0;
    }
    e(r,0)=e(r,0)/k(r,r);
    k(r,r)=1;

    if(r==0)
      break;
    r--;
  }

/*
** Ergebnisvektor zurückliefern
*/
```

Listing 637: Die Methode calcSolutionVector von CMathMatrix

```
  return(e);
}
```

Listing 637: Die Methode calcSolutionVector von CMathMatrix (Forts.)

Die Klasse `CMathMatrix` finden Sie auf der CD in der *CMathMatrix.h*-Datei.

Verschiedenes

138 Wie kann Speicher einfacher verwaltet werden?

Die Tatsache, dass es in C++ in der Verantwortung des Programmierers liegt, dynamisch angelegten Speicher auch wieder freizugeben, mag manchmal von Vorteil sein, häufig ist es aber einfach nur lästig.

Wir implementieren daher ein kleines Template `CMemory`, welches für eine bestimmte Anzahl von Objekten Speicher in Form eines Feldes reserviert.

Wird das `CMemory`-Objekt (z.B. durch Verlassen des Bezugsrahmens) zerstört, gibt es den Speicher automatisch frei, ohne dass sich der Programmier darum kümmern muss.

Natürlich könnte dieses Problem auch gelöst werden, indem eine Container-Klasse der STL benutzt wird, allerdings ist dies mit einem gewissen Overhead verbunden.

CMemory

```
template <typename Type>
class CMemory {
  private:
    Type *m_memory;

  public:
  CMemory(unsigned long anz) {
    m_memory=new(Type[anz]);
    if(!m_memory)
      throw(bad_alloc("CMemory::CMemory"));
  }

  ~CMemory()  {
     delete[](m_memory);
  }

  operator Type*() {
    return(m_memory);
  }
};
```

Listing 638: Das Template CMemory

139 Wie wird ein Objekt erzeugt, das seinen Wert zyklisch ändert?

Gefragt ist nach einem Objekt, welches beispielsweise die Werte 5, 7 und 11 zyklisch zur Verfügung stellt. Nehmen wir einmal folgendes Code-Fragment:

```
for(int x=0; x<10; x++)
  cout << s++ << " ";
```

Dann wäre die Ausgabe:

```
5 7 11 5 7 11 5 7 11 5
```

Falls Sie sich fragen, wozu so eine Funktionalität gut sein soll, dann sollten Sie einen kurzen Blick auf Rezept 140 werfen.

CShifter

Wir implementieren dazu eine Klasse `CShifter`, deren Objekte die gewünschten Eigenschaften besitzen.

Klassendefinition

Die Klasse besitzt nur zwei Attribute: einen Container, der die Werte zur Zyklenbildung aufnimmt (`m_elements`), und einen `long`-Wert für die aktuelle Position (`m_pos`.)

```
class CShifter {
private:
  std::vector<Type> m_elements;
  long m_pos;
};
```

Listing 639: Die Klassendefinition von CShifter

Konstruktoren

Zunächst einmal implementieren wir einen Kopier-Konstruktor:

```
CShifter(const CShifter &s)
  : m_pos(s.m_pos), m_elements(s.m_elements)
{}
```

Listing 640: Der Kopier-Konstruktor von CShifter

Zusätzlich wollen wir jeweils einen Konstruktor für die besonderen Fälle implementieren, dass der Zyklus nur aus zwei oder drei Werten besteht.

```
CShifter(const Type &a, const Type &b)
: m_pos(0) {
  m_elements.push_back(a);
  m_elements.push_back(b);
}

CShifter(const Type &a, const Type &b, const Type &c)
: m_pos(0) {
  m_elements.push_back(a);
  m_elements.push_back(b);
  m_elements.push_back(c);
}
```

Listing 641: Weitere Konstruktoren für CShifter

Zum Schluss fügen wir noch einen Konstruktor hinzu, der ein `CShifter`-Objekt aus einem Container erzeugt. Der Container muss dazu Iteratoren unterstützen:

```
template<typename CType>
CShifter(const CType &c)
: m_pos(0) {
  CType::const_iterator iter=c.begin();
  while(iter!=c.end())
    m_elements.push_back(*iter++);

/*
```

Listing 642: Ein CShifter-Konstruktor für Container

```
** Nicht mehr als ein Objekt enthalten*
** => Ausnahme werfen
*/
  if(m_elements.size()<2)
    throw std::invalid_argument("CShifter::CShifter()");
}
```

Listing 642: Ein CShifter-Konstruktor für Container (Forts.)

Inkrement-/Dekrement-Operatoren

Damit ein `CShifter`-Objekt seine Werte zyklisch durchlaufen kann, benötigt die Klasse natürlich entsprechende Inkrement- und Dekrement-Operatoren:

```
CShifter operator++(int) {
  CShifter tmp=*this;
  ++m_pos;
  if(m_pos>=m_elements.size())
    m_pos=0;
  return(tmp);
}

//*********

CShifter &operator++() {
  ++m_pos;
  if(m_pos>=m_elements.size())
    m_pos=0;
  return(*this);
}

//*********

CShifter operator--(int) {
  CShifter tmp=*this;
  if(m_pos==0)
    m_pos=m_elements.size()-1;
  else
    --m_pos;
  return(tmp);
}
```

Listing 643: Die Inkrement-/Dekrement-Operatoren von CShifter

```
//*********

CShifter &operator--() {
  if(m_pos==0)
    m_pos=m_elements.size()-1;
  else
    --m_pos;
  return(*this);
}
```

Listing 643: Die Inkrement-/Dekrement-Operatoren von CShifter (Forts.)

Rechenoperatoren

Damit nicht immer nur zum nächsten oder vorigen Wert gewechselt werden kann, wird die Klasse mit Additions- und Subtraktionsoperatoren ausgestattet.

```
CShifter &operator+=(long offset) {
  m_pos+=offset;
  if(m_pos>=m_elements.size())
    m_pos%=m_elements.size();
  return(*this);
}

//*********

CShifter &operator-=(long offset) {
  m_pos-=offset;
  if(m_pos<0)
    m_pos=m_elements.size()-1+
          ((m_pos+1)%static_cast<long>(m_elements.size()));
  return(*this);
}
```

Listing 644: Die Rechenoperatoren von CShifter

Zugriffsoperatoren

Damit auf den entsprechenden Wert einfacher zugegriffen werden kann, überladen wir unter anderem die `operator*`- und die `operator->`-Methode:

```
Type &operator*() {
  return(m_elements[m_pos]);
}

//*********

Type *operator->() {
  return(&m_elements[m_pos]);
}

//*********

operator Type() {
  return(m_elements[m_pos]);
}
```

Listing 645: Zugriffsoperatoren von CShifter

Verschiedene Methoden

Mit den Methoden `toBegin` und `toEnd` kann ein `CShifter`-Objekt auf das erste oder letzte Element des Zyklus gesetzt werden.

```
void toBegin() {
  m_pos=0;
}

//*********

void toEnd() {
  m_pos=m_elements.size()-1;
}
```

Listing 646: Die Methoden toBegin und toEnd von CShifter

Iterator

Eine STL-konforme Klasse ist immer vielseitiger einsetzbar, deswegen soll unsere `CShifter`-Klasse einen Iterator bekommen.

Klassendefinition

Die Klassendefinition des Iterators steht im öffentlichen Teil der Klasse `CShifter`.

```
class iterator;
friend class iterator;
class iterator
: public std::iterator<std::bidirectional_iterator_tag,Type> {
private:
  long m_pos;
  CShifter *m_shifter;
};
```

Konstruktoren

```
iterator()
  :m_shifter(0), m_pos(0)
{}

//*********

iterator(CShifter &shifter, long pos)
  : m_shifter(&shifter), m_pos(pos)
{}
```

Listing 647: Die Konstruktoren des Iterators

Inkrement-/Dekrement-Operatoren

Die Inkrement- und Dekrement-Operatoren ähneln stark denen der Klasse `CShifter`.

```
iterator &operator++() {
  ++m_pos;
  if(m_pos>=m_shifter->m_elements.size())
    m_pos=0;
  return(*this);
}
```

Listing 648: Die Inkrement-/Dekrement-Operatoren des Iterators

```
//*********

iterator &operator--() {
  if(m_pos==0)
    m_pos=m_shifter->m_elements.size()-1;
  else
    --m_pos;
  return(*this);
}

//*********

iterator operator++(int) {
  iterator tmp=*this;
  ++m_pos;
  if(m_pos>=m_shifter->m_elements.size())
    m_pos=0;
  return(tmp);
}

//*********

iterator operator--(int) {
  iterator tmp=*this;
  if(m_pos==0)
    m_pos=m_shifter->m_elements.size()-1;
  else
    --m_pos;
  return(tmp);
}
```

Listing 648: Die Inkrement-/Dekrement-Operatoren des Iterators (Forts.)

Vergleichsoperatoren

Ein bidirektionaler Iterator muss die Vergleichsoperatoren == und != zur Verfügung stellen:

```
bool operator==(iterator &i) {
  return((m_shifter==i.m_shifter)&&(m_pos==i.m_pos));
}
```

Listing 649: Die Vergleichsoperatoren des Iterators

```
//*********

bool operator!=(iterator &i) {
  return((m_shifter!=i.m_shifter)||(m_pos!=i.m_pos));
}
```

Listing 649: Die Vergleichsoperatoren des Iterators (Forts.)

Zugriffsoperatoren

Die Kategorie, der unser Iterator angehört, verlangt die Methoden `operator*` und `operator->`.

```
Type &operator*() {
  return(m_shifter->m_elements[m_pos]);
}

//*********

Type *operator->() {
  return(&m_shifter->m_elements[m_pos]);
}
```

Listing 650: operator und operator-> des Iterators*

Damit wir von einem `CShifter`-Objekt auch einen Iterator erzeugen können, implementieren wir noch die Methode `begin`. Eine `end`-Methode macht bei dieser Klasse keinen Sinn, weil die Werte zyklisch durchlaufen.

```
iterator begin() {
  return(iterator(*this,0));
}
```

Listing 651: Die Methode begin von CShifter

Sie finden die Klasse `CShifter` auf der CD in der *CShifter.h*-Datei.

140 Wie kann ein deutscher Personalausweis auf Echtheit geprüft werden?

Das Prüfen der Echtheit beschränkt sich hier auf die unterste Zeile des Personalausweises. In dieser unteren Zeile finden sich als Informationen die Seriennummer des Ausweises, das Geburtsdatum des Besitzers und das Ablaufdatum des Ausweises. Darüber hinaus sind noch diverse Prüfziffern enthalten, mit denen mehr oder weniger sichergestellt ist, dass die Daten nicht verändert wurden.

Die Auswertung dieser unteren Zeile findet im Internet immer mehr Verbreitung zur Kontrolle des Alters. Gibt man die untere Zeile an, kann man das Alter der Person in Erfahrung bringen und über die Prüfziffern feststellen, ob das Alter verändert wurde.

Schauen wir uns einmal die unterste Zeile eines fiktiven Ausweises an:

```
1234567897D<<7711226<0302014<<<<<<8
```

Die ersten zehn Ziffern bilden die Seriennummer des Ausweises inklusive einer Prüfziffer (in diesem Fall Seriennummer 123456789 mit Prüfziffer 7.)

Das darauf folgende »D« kennzeichnet die deutsche Staatsbürgerschaft (welch eine Überraschung bei einem deutschen Ausweis).

Der zweite Block hinter »<<« kodiert das Geburtsdatum des Besitzers in umgekehrter Reihenfolge mit einer Prüfziffer (in diesem Fall der 22.11.1977 mit Prüfziffer 6).

Der dritte Block hinter dem »<« steht für das Ablaufdatum des Ausweises mit Prüfziffer (hier der 01.02.03 mit Prüfziffer 4)

Die letzte Zahl der Zeile (hier 8) ist wieder eine Prüfziffer, dieses Mal über alle Ziffern der Zeile.

Eine Prüfziffer wird nun berechnet, indem die dazugehörigen Ziffern beim Durchlaufen abwechselnd mit 7, 3 oder 1 multipliziert werden. Die Ergebnisse werden aufaddiert und schließlich mit Modulo 10 auf eine Ziffer reduziert; die Prüfziffer. Abbildung 46 stellt den Sachverhalt am Beispiel der Seriennummer grafisch dar.

Für die Überprüfung eines Ausweises implementieren wir die Klasse `CGermanIdCard`, die die nötige Funktionalität beinhaltet.

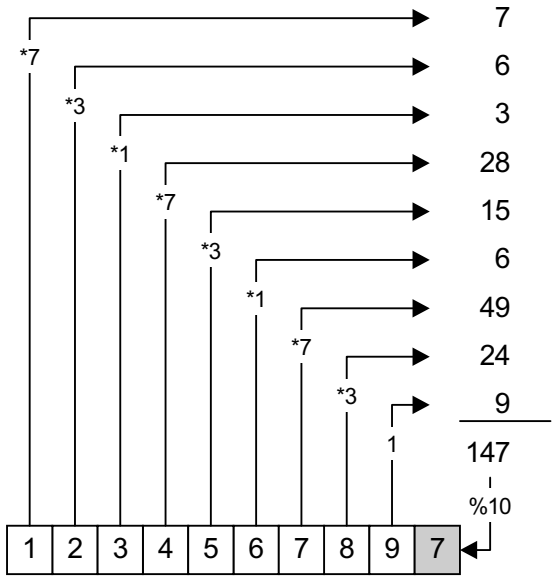


Abbildung 46: Die Berechnung der Prüfziffer

Klassendefinition

Die Klasse besitzt nur ein Attribut: einen String, in dem die untere Zeile des Ausweises gespeichert wird. Zusätzlich gibt es noch eine Zugriffsmethode, über die die Zeile ausgelesen werden kann.

```
class CGermanIdCard {
private:
  std::string m_id;
public:
  std::string getId() {
    return(m_id);
  }
};
```

Listing 652: Die Klassendefinition von CGermanIdCard

Konstruktoren

Es gibt zwei Konstruktoren. Dem ersten wird die untere Ausweiszeile als String übergeben. Der zweite erwartet die wesentlichen Teile der unteren Ausweiszeile in vier Strings.

```
CGermanIdCard::CGermanIdCard(const std::string &id) {

/*
** Wesentliche Bereiche isolieren
*/
  string serial=id.substr(0,10);
  string birth=id.substr(13,7);
  string expires=id.substr(21,7);
  string check=id.substr(34,1);

/*
** Bestehen Teile nur aus Zahlen?
*/
  if((!isNumericString(serial))||(!isNumericString(birth))||
     (!isNumericString(expires))||(!isNumericString(check)))
    throw invalid_argument("CGermanIdCard::CGermanIdCard");

/*
** Unterste Ausweiszeile zusammensetzen
*/
  m_id=serial+"D<<"+birth+"<"+expires+"<<<<<<"+check;

  if(id.size()!=35)
    throw invalid_argument("CGermanIdCard::CGermanIdCard");
  m_id=id;
}

//*********

CGermanIdCard::CGermanIdCard(const std::string &serial,
                             const std::string &birth,
                             const std::string &expires,
                             const std::string &check) {

/*
** Längen der einzelnen Teilbereiche überprüfen
*/
  if((serial.size()!=10)||(birth.size()!=7)||
     (expires.size()!=7)||(check.size()!=1))
    throw invalid_argument("CGermanIdCard::CGermanIdCard");
```

Listing 653: Die Konstruktoren von CGermanIdCard

```
/*
** Bestehen Teile nur aus Zahlen?
*/
  if((!isNumericString(serial))||(!isNumericString(birth))||
     (!isNumericString(expires))||(!isNumericString(check)))
    throw invalid_argument("CGermanIdCard::CGermanIdCard");

/*
** Unterste Ausweiszeile zusammensetzen
*/
  m_id=serial+"D<<"+birth+"<"+expires+"<<<<<<"+check;
  if(m_id.size()!=35)
    throw invalid_argument("CGermanIdCard::CGermanIdCard");
}
```

Listing 653: Die Konstruktoren von CGermanIdCard (Forts.)

Die Konstruktoren machen von der Funktion `isNumericString` aus Rezept 31 Gebrauch.

checkChecksum

Die statische Methode `checkChecksum` ist das Herzstück der Überprüfung. Ihr wird ein String übergeben, der die zu prüfenden Ziffern mitsamt ihrer Prüfziffer beinhaltet.

Sie berechnet dann aus den Ziffern die richtige Prüfziffer und vergleicht sie mit der angegebenen Prüfziffer am Ende des Strings.

Je nachdem, ob eine Übereinstimmung vorliegt, liefert die Methode einen booleschen Wert zurück.

```
bool CGermanIdCard::checkChecksum(const string &s) {
  CShifter<int> shifter(7,3,1);

  int chk=0;

/*
** Jede Ziffer mit entsprechendem Shifter-Wert
** multiplizieren und Ergebnisse aufaddieren
** Ergebnis Modulo 10 mit Prüfziffer vergleichen
```

Listing 654: Die Methode checkChecksum von CGermanIdCard

```
*/
  for(unsigned int i=0; i<s.size()-1; ++i)
    chk+=atoi(s.substr(i,1).c_str())*shifter++;
  return((chk%10)==atoi(s.substr(s.size()-1,1).c_str()));
}
```

Listing 654: Die Methode checkChecksum von CGermanIdCard (Forts.)

isIdValid

Die Methode `isIdValid` nimmt die tatsächliche Überprüfung der einzelnen Teile vor und liefert einen booleschen Wert zurück, der Auskunft darüber gibt, ob die untere Ausweiszeile korrekt ist oder nicht.

```
bool CGermanIdCard::isIdValid() {

/*
** Einzelne zu prüfende Teile extrahieren
*/
  string serial=m_id.substr(0,10);
  string birth=m_id.substr(13,7);
  string expiration=m_id.substr(21,7);

/*
** Relevante Teile für gesamte Prüfziffer zusammensetzen
*/
  string compID=serial+birth+expiration+m_id.substr(34,1);

/*
** Prüfziffern überprüfen
*/
  return(checkChecksum(serial)&&
         checkChecksum(birth)&&
         checkChecksum(expiration)&&
         checkChecksum(compID));
}
```

Listing 655: Die Methode isIdValid von CGermanIdCard

Sie finden die Klasse `CGermanIdCard` auf der CD in den `CGermanIdCard`-Dateien.

141 Wie kann das Geburtsdatum aus den Personalausweis-Daten ermittelt werden?

Das Geburtsdatum ist in den Ausweisdaten kodiert. Eine Unschärfe ergibt sich allerdings aus der Tatsache, dass das Geburtsjahr nur zweistellig angegeben ist.

Wir unterstellen in unserer Berechnung, dass Personen unter zehn Jahren keinen eigenen Personalausweis besitzen.

getAge

Wir erweitern die Klasse `CGermanIdCard` um die Funktion `getAge`, die das Alter des Besitzers bestimmt.

```
CDate CGermanIdCard::getAge() {

/*
** Erst einmal davon ausgehen, dass Person in
** diesem Jahrhundert geboren wurde
*/
  CDate date(atoi(m_id.substr(17,2).c_str()),
             atoi(m_id.substr(15,2).c_str()),
             atoi(m_id.substr(13,2).c_str())+2000);

/*
** Ist Person jünger als 10 oder noch
** gar nicht geboren?
** => Person muss im vorigen Jahrhundert geboren worden sein
*/
  if(CDate::calcAge(date)<10)
    date.setYear(date.getYear()-100);

  return(date);
}
```

Listing 656: Die Methode getAge von CGermanIdCard

Die Methode greift auf die `CDate`-Klasse aus Rezept 48 zurück.

Sie finden die Klasse `CGermanIdCard` auf der CD in den `CGermanIdCard`-Dateien.

142 Wie kann aus den Personalausweis-Daten die Volljährigkeit bestimmt werden?

Mit der Methode `getAge` aus Rezept 141 und der Methode `calcAge` aus Rezept 62 ist diese Frage leicht beantwortet.

isOfAge

Wir implementieren dazu die `CGermanIdCard`-Methode `isOfAge`:

```
bool CGermanIdCard::isOfAge() {
  return(CDate::calcAge(getAge())>=18);
}
```

Listing 657: Die Methode isOfAge von CGermanIdCard

Sie finden die Klasse `CGermanIdCard` auf der CD in den `CGermanIdCard`-Dateien.

143 Wie kann aus den Personalausweis-Daten das Ablaufdatum bestimmt werden?

Auch bei dem Ablaufdatum ist das Jahr im Ausweis nur als zweistellige Zahl kodiert. Da ein Ausweis aber nie länger als zehn Jahre gültig ist, lässt sich aus einem länger gültigen Ausweis schließen, dass er bereits im letzten Jahrhundert abgelaufen ist.

getExpirationDate

Wir fügen zur Klasse `CGermanIdCard` die Methode `getExpirationDate` hinzu, die das Ablaufdatum extrahiert.

```
CDate CGermanIdCard::getExpirationDate() {

/*
** Erst einmal davon ausgehen, dass Ausweis
** in diesem Jahrhundert abläuft
*/
  CDate date(atoi(m_id.substr(25,2).c_str()),
             atoi(m_id.substr(23,2).c_str()),
             atoi(m_id.substr(21,2).c_str())+2000);
```

Listing 658: Die Methode getExpirationDate von CGermanIdCard

```
/*
** Ist Ausweis noch länger als 10 Jahre gültig?
** => Ausweis muss bereits im letzten Jahrhundert abgelaufen sein
*/
  CDate today;
  if(date.getYear()-today.getYear()>10)
    date.setYear(date.getYear()-100);
  return(date);
}
```

Listing 658: Die Methode getExpirationDate von CGermanIdCard (Forts.)

Sie finden die Klasse `CGermanIdCard` auf der CD in den `CGermanIdCard`-Dateien.

144 Wie kann aus den Personalausweis-Daten ermittelt werden, ob der Ausweis noch gültig ist?

Diese Frage ist mit der Methode aus dem letzten Rezept leicht zu beantworten.

isExpired

```
bool CGermanIdCard::isExpired() {
  CDate today;
  return(getExpirationDate()<today);
}
```

Listing 659: Die Methode isExpired von CGermanIdCard

Sie finden die Klasse `CGermanIdCard` auf der CD in den `CGermanIdCard`-Dateien.

145 Wie kann ein deutscher Reisepass auf Echtheit geprüft werden?

Genau wie beim Personalausweis beschränkt sich unsere Prüfmethode auf die letzte Zeile des Plastikteils im Reisepass. Diese Zeile ist ähnlich aufgebaut wie die untere Zeile des Personalausweises, hat jedoch eine unterschiedliche Länge und zusätzliche Informationen.

Die Prüfsummenberechnung allerdings ist mit der des Personalausweises identisch. Eine grafische Erklärung finden Sie in Abbildung 46.

Wir werden daher auf die statische Methode `checkChecksum` von `CGermanIdCard` aus Rezept 140 zurückgreifen.

Klassendefinition

Die Klasse besitzt als einziges Attribut einen String für die untere Zeile des Reisepasses.

```
class CGermanPassport {
private:
  std::string m_id;
public:
  std::string getId() {
    return(m_id);
  }
};
```

Listing 660: Die Klassendefinition von CGermanPassport

Konstruktoren

Es existieren zwei Konstruktoren. Dem ersten wird die komplette untere Zeile als ein String übergeben. Der zweite besitzt als Parameter die Einzelteile der unteren Zeile und setzt diese entsprechend zusammen.

```
CGermanPassport::CGermanPassport(const std::string &id) {

/*
** Wesentliche Bereiche isolieren
*/
  string serial=id.substr(0,10);
  string birth=id.substr(13,7);
  string sex=id.substr(20,1);
  string expires=id.substr(21,7);
  string check=id.substr(34,1);

/*
** Bestehen Teile nur aus Zahlen?
*/
  if((!isNumericString(serial))||(!isNumericString(birth))||
     (!isNumericString(expires))||(!isNumericString(check)))
```

Listing 661: Die Konstruktoren von CGermanPassport

```
    throw invalid_argument("CGermanPassport::CGermanPassport");

/*
** Gültige Geschlechtsangabe?
*/
  if((sex!="M")&&(sex!="F"))
    throw invalid_argument("CGermanPassport::CGermanPassport");

/*
** Unterste Passzeile zusammensetzen
*/
  m_id=serial+"D<<"+birth+sex+expires+"<<<<<<<<<<<<<<<"+check;

  if(id.size()!=44)
    throw invalid_argument("CGermanPassport::CGermanPassport");
  m_id=id;
}

//*********

CGermanPassport::CGermanPassport(const std::string &serial,
                                 const std::string &birth,
                                 const std::string &sex,
                                 const std::string &expires,
                                 const std::string &check) {

/*
** Längen der einzelnen Teilbereiche überprüfen
*/
  if((serial.size()!=10)||(birth.size()!=7)||
     (expires.size()!=7)||(check.size()!=1))
    throw invalid_argument("CGermanPassport::CGermanPassport");

/*
** Bestehen Teile nur aus Zahlen?
*/
  if((!isNumericString(serial))||(!isNumericString(birth))||
     (!isNumericString(expires))||(!isNumericString(check)))
    throw invalid_argument("CGermanPassport::CGermanPassport");

/*
```

Listing 661: Die Konstruktoren von CGermanPassport (Forts.)

```
** Gültige Geschlechtsangabe?
*/
  if((sex!="M")&&(sex!="F"))
    throw invalid_argument("CGermanPassport::CGermanPassport");

/*
** Unterste Passzeile zusammensetzen
*/
  m_id=serial+"D<<"+birth+sex+expires+"<<<<<<<<<<<<<<"+check;
  if(m_id.size()!=44)
    throw invalid_argument("CGermanPassport::CGermanPassport");
}
```

Listing 661: Die Konstruktoren von CGermanPassport (Forts.)

Die Konstruktoren machen von der Funktion `isNumericString` aus Rezept 31 Gebrauch.

isIdValid

Die Methode `isIdValid` prüft, ob die Prüfziffern der untersten Zeile korrekt sind.

```
bool CGermanPassport::isIdValid() {

/*
** Einzelne zu prüfende Teile extrahieren
*/
  string serial=m_id.substr(0,10);
  string birth=m_id.substr(13,7);
  string expiration=m_id.substr(21,7);

/*
** Relevante Teile für gesamte Prüfziffer zusammensetzen
*/
  string compID=serial+birth+expiration+m_id.substr(34,1);

/*
** Prüfziffern überprüfen
*/
  return(CGermanIdCard::checkChecksum(serial)&&
         CGermanIdCard::checkChecksum(birth)&&
```

Listing 662: Die Methode isIdValid von CgermanPassport

```
        CGermanIdCard::checkChecksum(expiration)&&
        CGermanIdCard::checkChecksum(compID));
}
```

Listing 662: Die Methode isIdValid von CgermanPassport (Forts.)

`isIdValid` verwendet die statische `CGermanIdCard`-Methode `checkChecksum` aus Rezept 140.

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

146 Wie kann das Geburtsdatum aus den Reisepass-Daten ermittelt werden?

Wie beim Personalausweis unterstellen wir, dass keine Person unter zehn Jahren einen eigenen Reisepass besitzt (obwohl das im Gegensatz zum Personalausweis durchaus möglich wäre).

getAge

Die Methode `getAge` berechnet das Alter des Pass-Besitzers.

```
CDate CGermanPassport::getAge() {

/*
** Erst einmal davon ausgehen, dass Person in
** diesem Jahrhundert geboren wurde
*/
  CDate date(atoi(m_id.substr(17,2).c_str()),
             atoi(m_id.substr(15,2).c_str()),
             atoi(m_id.substr(13,2).c_str())+2000);

/*
** Ist Person jünger als 10 oder noch
** gar nicht geboren?
** => Person muss im vorigen Jahrhundert geboren worden sein
*/
  if(CDate::calcAge(date)<10)
    date.setYear(date.getYear()-100);
```

Listing 663: Die Methode getAge von CGermanPassport

```
  return(date);
}
```

Listing 663: Die Methode getAge von CGermanPassport (Forts.)

Die Methode greift auf die `CDate`-Klasse aus Rezept 48 zurück.

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

147 Wie kann aus den Reisepass-Daten die Volljährigkeit bestimmt werden?

Wir greifen dazu wieder auf die Methoden `getAge` aus Rezept 141 und `calcAge` aus Rezept 62 zurück.

isOfAge

```
bool CGermanPassport::isOfAge() {
  return(CDate::calcAge(getAge())>=18);
}
```

Listing 664: Die Methode isOfAge von CGermanPassport

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

148 Wie kann aus den Reisepass-Daten das Ablaufdatum bestimmt werden?

Das Ablaufdatum des Reisepasses ist ebenfall nur mit einer zweistelligen Jahreszahl kodiert. Da auch ein Reisepass nie länger als zehn Jahre gültig ist, lässt sich aus einem länger gültigen Reisepass wieder schließen, dass er bereits im letzten Jahrhundert abgelaufen ist.

getExpirationDate

Wir fügen zur Klasse `CGermanIdCard` die Methode `getExpirationDate` hinzu, die das Ablaufdatum extrahiert.

```
CDate CGermanPassport::getExpirationDate() {

/*
** Erst einmal davon ausgehen, dass der Pass
** in diesem Jahrhundert abläuft
*/
  CDate date(atoi(m_id.substr(25,2).c_str()),
             atoi(m_id.substr(23,2).c_str()),
             atoi(m_id.substr(21,2).c_str())+2000);

/*
** Ist der Pass noch länger als 10 Jahre gültig?
** => Pass muss bereits im letzten Jahrhundert abgelaufen sein
*/
  CDate today;
  if(date.getYear()-today.getYear()>10)
    date.setYear(date.getYear()-100);
  return(date);
}
```

Listing 665: Die Methode getExpirationDate von CGermanPassport

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

149 Wie kann aus den Reisepass-Daten ermittelt werden, ob der Pass noch gültig ist?

Diese Frage ist mit der Methode aus dem letzten Rezept leicht zu beantworten.

isExpired

```
bool CGermanPassport::isExpired() {
  CDate today;
  return(getExpirationDate()<today);
}
```

Listing 666: Die Methode isExpired von CGermanPassport

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

150 Wie kann aus den Reisepass-Daten das Geschlecht des Besitzers ermittelt werden?

Wir ergänzen dazu die Klasse `CGermanPassport` um die Methode `isMale`, die einen wahren Wert zurückliefert, wenn der Besitzer männlich ist, und um die Methode `isFemale`, die bei einem weiblichen Besitzer einen wahren Wert zurückliefert.

```
bool CGermanPassport::isMale() {
  return(m_id.substr(20,1)=="M");
}

//*********

bool CGermanPassport::isFemale() {
  return(m_id.substr(20,1)=="F");
}
```

Listing 667: Die Methoden isMale und isFemale von CGermanPassport

Sie finden die Klasse `CGermanPassport` auf der CD in den `CGermanPassport`-Dateien.

151 Wie können Laufzeiten gemessen werden?

Häufig findet man für ein Problem mehrere Lösungsansätze. Und dann ist die Frage, welcher Ansatz der effizientere ist.

Das kann bei ganz trivialen Fragen beginnen, zum Beispiel, ob bei einer Traversion der Zugriff über Iteratoren oder über den Index-Operator geschehen soll.

Aber nicht immer sind die Antworten offensichtlich, deswegen bietet es sich im Zweifelsfall an, die zur Debatte stehenden Ansätze jeweils x-mal hintereinander laufen zu lassen und anschließend die Laufzeiten zu vergleichen.

CStopwatch

Wir wollen dazu eine Art Stoppuhr programmieren und kapseln die Funktionalität in der Klasse `CStopwatch`.

Zur Ermittlung einer genauen Zeit taugt die in Rezept 48 vorgestellte `tm`-Struktur nur bedingt, weil ihre Genauigkeit nur im Sekundenbereich liegt.

Wir benutzen stattdessen die Funktion `clock`, die die seit dem Start des Programms vergangenen Clocks in Form eines `clock_t`-Objektes zurückgibt.

Wie viele Clocks eine Sekunde ergeben, liefert die Konstante `CLOCKS_PER_SEC`. Im Normalfall ist ein Clock genau eine Millisekunde.

Klassendefinition

Das Attribut `m_running` hilft zu verhindern, dass ein bereits gestartetes Objekt nochmals gestartet bzw. ein schon gestopptes Objekt erneut gestoppt wird.

In `m_start` wird die Startzeit festgehalten, um später beim Stoppen die Differenz der beiden Zeiten bilden zu können.

Die vom Objekt erfasste Zeit wird im `CTimevector`-Objekt `m_elapsed` gespeichert.

```
class CStopwatch {
private:
  bool m_running;
  clock_t m_start;
  CTimevector m_elapsed;
};
```

Listing 668: Die Klassendefinition von CStopwatch

Zugriffsmethoden

Die einzelnen Teile der gemessenen Zeit sollen ermittelbar sein, deswegen implementieren wir für sie Zugriffsmethoden, die sich zum Teil der Funktionalität der `CTime`-Klasse aus Rezept 59 bedienen.

```
CTimevector getElapsed() const {
  return(m_elapsed);
}

int getDays() const {
  return(m_elapsed.getDays());
}

int getHours() const {
  return(CTime(m_elapsed.getMilliseconds()).getHour());
}

int getMinutes() const {
  return(CTime(m_elapsed.getMilliseconds()).getMinute());
```

Listing 669: Die Zugriffsmethoden von CStopwatch

```
}

int getSeconds() const {
  return(CTime(m_elapsed.getMilliseconds()).getSecond());
}

int getMilliseconds() const {
  return(CTime(m_elapsed.getMilliseconds()).getMillisecond());
}
```

Listing 669: Die Zugriffsmethoden von CStopwatch (Forts.)

getDHHMMSSMMM

Um direkt ein ausgabefähiges Format zu erhalten, fügen wir eine Methode `getDHHMMSMMM` hinzu:

```
string CStopwatch::getDHHMMSSMMM() const {

  stringstream str;
  CTime time(m_elapsed.getMilliseconds());

  str.fill('0');
  str << m_elapsed.getDays() << "D ";
  str << setw(2) << time.getHour() << ":";
  str << setw(2) << time.getMinute() << ":";
  str << setw(2) << time.getSecond() << ":";
  str << setw(3) << time.getMillisecond();// << "M";
  return(str.str());
}
```

Listing 670: Die Methode getDHHMMSSMMM von CStopwatch

start

Mit der Methode `start` wird die Zeiterfassung gestartet.

```
bool CStopwatch::start() {
  if(m_running)
    return(false);
```

Listing 671: Die Methode start von CStopwatch

```
  m_start=clock();
  m_running=true;
  return(true);
}
```

Listing 671: Die Methode start von CStopwatch (Forts.)

stop

Die Methode `stop` nun hält die aktuelle Zeiterfassung an und ermittelt die vergangene Zeit. Da die neu ermittelte Zeit auf `m_elapsed` aufaddiert wird, können `start` und `stop` mehrmals paarweise aufgerufen werden, um anschließend eine Gesamtzeit zu erhalten.

Wegen des Wertebereichs von `unsigned long` dürfen zwischen dem `start`- und dem `stop`-Aufruf nicht mehr als 49 Tage liegen. Das sollte eigentlich die üblichen Testzeiten abdecken.

Die von `CStopwatch` gemessene Gesamtzeit darf allerdings mehrere Tausend Jahre betragen, wobei die obere Leistungsgrenze der Klasse eher selten ausgereizt werden dürfte.

```
bool CStopwatch::stop() {
  if(!m_running)
    return(false);
  unsigned long elapsed=static_cast<unsigned long>(
                        (clock()-m_start)*
                        (1000.0/CLOCKS_PER_SEC));
  m_elapsed=m_elapsed+
            CTimevector(elapsed/86400000,elapsed%86400000);
  m_running=false;
  return(true);
}
```

reset

Die `reset`-Methode setzt die von `CStopwatch` erfasste Zeit wieder auf Null zurück.

```
void CStopwatch::reset() {
```

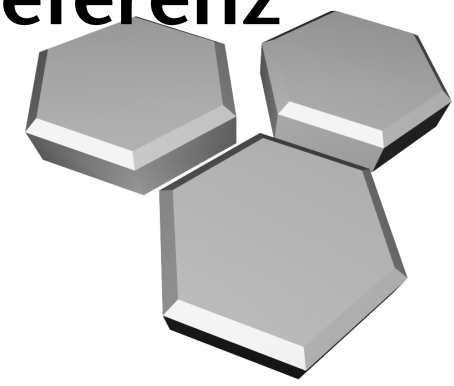
Listing 672: Die Methode reset von CStopwatch

```
  m_elapsed=CTimevector();
}
```

Listing 672: Die Methode reset von CStopwatch (Forts.)

Sie finden die Klasse `CStopwatch` auf der CD in den `CStopwatch`-Dateien.

Teil III
Referenz



Referenz

In dieser Referenz sind alle wichtigen Funktionen, Algorithmen und Begriffe aufgeführt und knapp erklärt.

acos

Funktion der Standardbibliothek, definiert in `cmath`.

Arcus Kosinus: die Umkehrfunktion des Kosinus für `a` im Bogenmaß im Bereich [0,pi].

```
double acos(double a);
```

adjacent_find

Algorithmus der STL, definiert in `algorithm`.

Sucht nach benachbarten Duplikaten.

Varianten:

```
template<class Forward>
Forward adjacent_find(Forward anf, Forward end);

template<class Forward, class BinPred>
Forward adjacent_find(Forward anf, Forward end, BinPred fkt);
```

asin

Funktion der Standardbibliothek, definiert in `cmath`.

Arcus Sinus: die Umkehrfunktion des Sinus für *a* im Bogenmaß im Bereich [-pi/2, pi/2].

```
double asin(double a);
```

atan

Funktion der Standardbibliothek, definiert in `cmath`.

Arcus Tangens: Die Umkehrfunktion des Tangens für `a` im Bogenmaß im Bereich [-pi/2, pi/2].

```
double atan(double a);
```

atan2

Funktion der Standardbibliothek, definiert in `cmath`.

Arcus Tangens: Die Umkehrfunktion des Tangens für `a/b` im Bogenmaß im Bereich [-pi, pi].

```
double atan2(double a, double b);
```

atof

Funktion der Standardbibliothek, definiert in `cstdlib`.

Liefert einen `double`-Wert, der dem Wert entspricht, den die Zeichen des Strings `s` darstellen. Die in `s` enthaltene Zahl kann jedes von C++ bei der Ausgabe von `double`-Werten erlaubte Format besitzen.

```
double atof(const char *s);
```

atoi

Funktion der Standardbibliothek, definiert in `cstdlib`.

Liefert einen `int`-Wert, der dem Wert entspricht, den die Zeichen des Strings `s` darstellen. Die in `s` enthaltene Zahl kann jedes von C++ bei der Ausgabe von `int`-Werten erlaubte Format besitzen.

```
int atoi(const char *s);
```

atol

Funktion der Standardbibliothek, definiert in `cstdlib`.

Liefert einen `long`-Wert, der dem Wert entspricht, den die Zeichen des Strings `s` darstellen. Die in `s` enthaltene Zahl kann jedes von C++ bei der Ausgabe von `long`-Werten erlaubte Format besitzen.

```
long atol(const char *s);
```

binary_search

Algorithmus der STL, definiert in `algorithm`.

Sucht in einem sortierten Bereich nach einem Objekt.

Varianten:

```
template<class Forward, class Typ>
bool binary_search(Forward anf, Forward end,
                   const Typ &obj);

template<class Forward, class Typ, class BinPred>
bool binary_search(Forward anf, Forward end,
                   const Typ &obj, BinPred fkt);
```

bool

Datentyp für boolesche Werte. Kann nur die Werte `true` oder `false` annehmen.

break

Springt aus dem innersten `for`-, `do`-, `while`- oder `switch`-Anweisungsblock heraus. Der Programmfluss geht hinter dem Anweisungsblock weiter.

case

Definiert innerhalb eines `switch`-Blocks eine Sprungmarke. Siehe auch: `switch`.

catch

`catch`-Blöcke stehen immer hinter einem `try`-Block. Mit einem `catch`-Block lässt sich eine innerhalb des dazugehörigen `try`-Blocks geworfene Ausnahme auffangen und bearbeiten.

ceil

Funktion der Standardbibliothek, definiert in `cmath`.

Liefert die nächsthöhere Ganzzahl von `a`: Aus 3.2 wird 4, aus –8.6 wird –8.

```
double ceil(double a);
```

char

Elementarer Datentyp, der ein Zeichen aufnehmen kann. Er besitzt üblicherweise einen Wertebereich von 0-255.

class

Schlüsselwort zur Definition einer Klasse.

const

Deklariert ein Objekt als konstant.

const_cast

Entfernt eine eventuelle `const`-Eigenschaft eines nicht konstanten Objekts.

```
const_cast<Typ>(Ausdruck)
```

continue

Springt zum Ende des innersten `for`-, `do`- oder `while`-Anweisungsblocks und leitet damit den nächsten Schleifenschritt ein.

copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden.

```
template<class Input, class Output>
Output copy(Input anf1, Input end1, Output anf2);
```

copy_backward

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden. Der Kopiervorgang beginnt mit dem letzten Objekt.

```
template<class Bi1, class Bi2>
Bi2 copy_backward(Bi1 anf1, Bi1 end1, Bi2 end2){
```

cos

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnung des Kosinus von `a` im Bogenmaß.

```
double cos(double a);
```

cosh

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnung des Kosinus Hyperbolicus von `a` im Bogenmaß.

```
double cosh(double a);
```

count

Algorithmus der STL, definiert in `algorithm`.

Zählt alle mit einem Objekt identischen Objekte.

Varianten:

```
template<class Input, class Typ>
iterator_traits<Input>::difference_type
count(Input anf, Input end, const Typ &obj);
```

count_if

Algorithmus der STL, definiert in `algorithm`.

Zählt alle Objekte, für die ein Funktionsaufruf einen wahren Wert liefert.

Varianten:

```
template<class Input, class Pred>
iterator_traits<Input>::difference_type
count_if(Input anf, Input end, Pred fkt);
```

do { } while

Ein Schleifenkonstrukt. Siehe Kapitel »Grundlagen« Rezept 6.

default

Definiert innerhalb eines `switch`-Blocks diejenige Sprungmarke, die immer dann angesprungen wird, wenn keine spezielle Sprungmarke definiert wurde. Siehe auch: `switch`.

delete

Löscht den an der übergebenen Adresse liegenden, dynamisch angeforderten Speicher. Siehe auch: `new`.

double

Mit Vorzeichen behafteter, elementarer Fließkomma-Datentyp. Mindestens so groß wie `float`, im Normalfall aber größer. Die genaue Größe ist implementierungsabhängig.

dynamic_cast

Wandelt einen Ausdruck in einen anderen Typ um. Es findet eine Typüberprüfung zur Laufzeit statt. Dazu müssen die Typinformationen zur Laufzeit vorliegen, sprich: Es muss sich um polymorphe Typen handeln.

```
dynamic_cast<Typ>(Ausdruck)
```

equal

Algorithmus der STL, definiert in `algorithm`.

Prüft die Objekte zweier Rezepte auf Gleichheit.

Varianten:

```
template<class Input1, class Input2>
bool equal(Input1 anf1, Input1 end1, Input2 anf2);

template<class Input1, class Input2, class BinPred>
bool equal(Input1 anf1, Input1 end1,
           Input2 anf2, BinPred fkt);
```

equal_range

Algorithmus der STL, definiert in `algorithm`.

Sucht in einem entsprechenden Bereich nach dem ersten Bereich, dessen Objekte gleich mit einem Objekt sind, und liefert die Grenzen des gefundenen Bereichs als Paar zurück.

Varianten:

```
template<class Forward, class Typ>
pair<Forward, Forward> equal_range(Forward anf, Forward end,
                                   const Typ &obj);

template<class Forward, class Typ, class BinPred>
pair<Forward, Forward> equal_range(Forward anf, Forward end,
                                const Typ &obj, BinPred fkt);
```

else

Alternativer Zweig einer `if`-Anweisung. Siehe Kapitel »Grundlagen« Rezept 6.

enum

Schlüsselwort zur Definition eines Aufzählungstyps.

```
enum Wochentage{mon,die,mit,don,fre,sam,son};
```

Der im oberen Beispiel selbst definierte Datentyp `Wochentage` kann nur die definierten Werte aufnehmen.

```
Wochentage x=mon;
```

exp

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnung des Exponentialwerts e^{α} von `a`.

```
double exp(double a);
```

explicit

Einparametrige Konstruktoren können vom Compiler zur impliziten Typumwandlung verwendet werden. Dies kann verhindert werden, indem der Konstruktor als `explicit` deklariert wird.

export

Schlüsselwort zur Aufteilung von Templates in *.h*- und *.cpp*-Datei.

extern

Schlüsselwort zur Deklaration einer an anderer Stelle definierten Variablen.

fabs

Funktion der Standardbibliothek, definiert in `cmath`.

Liefert den Betrag bzw. Absolutwert von `a`.

```
double fabs(double a);
```

false

Einer der Werte, die der boolesche Datentyp `bool` annehmen kann. `false` steht für »falsch« und besitzt normalerweise den numerischen Wert 0.

fill

Algorithmus der STL, definiert in `algorithm`.

Ersetzt alle Elemente eines Bereichs durch dasselbe Objekt.

```
template<class Forward, class Typ>
void fill(Forward anf, Forward end, const Typ &nwert);
```

fill_n

Algorithmus der STL, definiert in `algorithm`.

Ersetzt ab einer entsprechenden Iterator-Position eine Anzahl von Objekten durch dasselbe Objekt.

```
template<class Output, class Anzahl, class Typ>
void fill_n(Output anf, Anzahl anz, const Typ &nwert);
```

find

Algorithmus der STL, definiert in `algorithm`.

Sucht nach dem ersten Vorkommen eines Objekts.

Varianten:

```
template<class Forward, Class Typ>
Forward find(Forward anf, Forward end, const Typ &obj);
```

find_end

Algorithmus der STL, definiert in `algorithm`.

Sucht in der ersten Sequenz nach dem letzten Vorkommen der zweiten Sequenz.

Varianten:

```
template<class Forward1, class Forward2>
Forward1 find_end(Forward1 anf1, Forward1 end1,
                  Forward2 anf2, Forward2 end2);

template<class Forward1, class Forward2, class BinPred>
Forward1 find_end(Forward1 anf1, Forward1 end1,
                 Forward2 anf2, Forward2 end2, BinPred fkt);
```

find_first_of

Algorithmus der STL, definiert in `algorithm`.

Sucht im ersten Bereich nach dem ersten Vorkommen eines Objekts aus dem zweiten Bereich.

Varianten:

```
template<class Forward1, class Forward2>
Forward1 find_first_of(Forward1 anf1, Forward1 end1,
                       Forward2 anf2, Forward2 end2);

template<class Forward1, class Forward2, class BinPred>
Forward1 find_first_of(Forward1 anf1, Forward1 end1,
               Forward2 anf2, Forward2 end2, BinPred fkt);
```

find_if

Algorithmus der STL, definiert in `algorithm`.

Sucht nach dem ersten Vorkommen eines Objekts, für das ein Funktionsaufruf einen wahren Wert liefert.

```
template<class Forward, class Pred>
Forward find_if(Forward anf, Forward end, Pred fkt);
```

float

Kleinster der mit Vorzeichen behafteten elementaren Fließkomma-Datentypen. Die genaue Größe ist implementierungsabhängig.

floor

Funktion der Standardbibliothek, definiert in `cmath`.

Liefert die nächstniedrigere Ganzzahl von `a`: Aus 3.2 wird 3, aus –8.6 wird –9.

```
double floor(double a);
```

fmod

Funktion der Standardbibliothek, definiert in `cmath`.

Liefert den Modulowert `a%b` bzw. den Rest der Division `a/b`.

```
double fmod(double a, double b);
```

for

Ein Schleifenkonstrukt. Siehe Kapitel »Grundlagen« Rezept 6.

for_each

Algorithmus der STL, definiert in `algorithm`.

Ruft für alle Objekte eines Bereichs ein Funktionsobjekt auf und liefert dieses anschließend zurück.

```
template<class Input, class Op>
Op for_each(Input anf, Input end, Op fkt);
```

frexp

Funktion der Standardbibliothek, definiert in `cmath`.

Die Fließkommazahl `a` wird in einen ganzzahligen Wert i und einen Fließkommawert f im Bereich [0.5, 1] aufgespalten, so dass $a=f*2^i$ gilt. i wird in `b` gespeichert und f von der Funktion zurückgegeben. Für `a` gleich 0 ist `b` ebenfalls 0.

```
double frexp(double a, int *b);
```

friend

Mit diesem Schlüsselwort werden innerhalb einer Klasse Funktionen oder Klassen deklariert, die auf private oder geschützte Elemente der die `friend`-Deklaration beinhaltenden Klasse/Struktur zugreifen dürfen.

generate

Algorithmus der STL, definiert in `algorithm`.

Ersetzt alle Elemente eines Bereichs durch den Rückgabewert eines Funktionsaufrufs.

```
template<class Forward, class Generator>
void generate(Forward anf, Forward end, Generator gen);
```

generate_n

Algorithmus der STL, definiert in `algorithm`.

Ersetzt ab einer entsprechenden Iterator-Position eine Anzahl von Elementen durch den Rückgabewert eines Funktionsaufrufs.

```
template<class Output, class Anzahl, class Generator>
void generate_n(Output anf, Anzahl anz, Generator gen);
```

HUGE_VAL

Konstante der Standardbibliothek, definiert in `cmath`.

Steht für den Rückgabewert mancher Funktionen bei einer Wertüberschreitung. Er kann sowohl positiv als auch negativ sein.

if

Ein Verzweigungs-Konstrukt. Siehe Kapitel »Grundlagen« Rezept 6.

includes

Algorithmus der STL, definiert in `algorithm`.

Sind alle Elemente der ersten Menge auch in der zweiten Menge enthalten, dann liefert die Funktion `true` zurück, andernfalls `false`.

Varianten:

```
template<class Input1, class Input2>
bool includes(Input1 anf1, Input1 end1,
              Input2 anf2, Input2 end2);

template<class Input1, class Input2, class BinPred>
bool includes(Input1 anf1, Input1 end1,
              Input2 anf2, Input2 end2, BinPred fkt);
```

inline

Mit diesem Schlüsselwort deklarierte Methoden/Funktionen werden nicht aufgerufen, sondern der Funktionsaufruf wird wie bei einem Makro durch den Programmtext der Funktion ersetzt.

`inline` ist eine Empfehlung, sie ist für den Compiler nicht zwingend. Zum Beispiel wird der Compiler normalerweise die `inline`-Deklaration bei Methoden/Funktionen, die Schleifen enthalten, ignorieren.

inplace_merge

Algorithmus der STL, definiert in `algorithm`.

Verschmilzt in einem Bereich den sortierten Bereich vor `mit` mit dem sortierten Bereich hinter `mit` zu einem gesamten sortierten Bereich.

Varianten:

```
template<class Bi>
void inplace_merge(Bi anf, Bi mit, Bi end);

template<class Bi, class BinPred>
void inplace_merge(Bi anf, Bi mit, Bi end, BinPred fkt);
```

int

Ganzzahliger, vorzeichenbehafteter, elementarer Datentyp. Mindestens 2 Bytes groß.

isalnum

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eins der Zeichen a-z, A-Z oder 0-9 ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int isalnum(int a);
```

isalpha

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eins der Zeichen a-z oder A-Z ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int isalpha(int a);
```

iscntrl

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eins der Zeichen FF, NL, CR, HT, VT, BEL oder BS ist.

```
int iscntrl(int a);
```

isdigit

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eine der Ziffern 0-9 ist.

```
int isdigit(int a);
```

isgraph

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` ein darstellbares Zeichen, aber kein Leerzeichen ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int isgraph(int a);
```

islower

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn *a* eins der Zeichen a-z ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int islower(int a);
```

isprint

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` ein darstellbares Zeichen oder ein Leerzeichen ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int isprint(int a);
```

ispunct

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `isgraph(a)` wahr und `isalnum(a)` falsch ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int ispunct(int a);
```

isspace

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` ein Leerzeichen, FF, NL, CR, HT oder VT ist

```
int isspace(int a);
```

isupper

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eins der Zeichen A-Z ist.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int isupper(int a);
```

isxdigit

Funktion der Standardbibliothek, definiert in `cctype`.

Liefert einen wahren Wert, wenn `a` eine hexadezimale Ziffer ist (0-9, a-f oder A-F).

```
int isxdigit(int a);
```

iter_swap

Algorithmus der STL, definiert in `algorithm`.

Vertauscht die Objekte an den beiden Iterator-Positionen.

```
template<class Forward1, class Forward2>
inline void iter_swap(Forward1 i1, Forward2 i2);
```

ldexp

Funktion der Standardbibliothek, definiert in `cmath`.

Umkehrfunktion von `frexp`: berechnet den Wert $a*2^{\beta}$.

```
double ldexp(double a, int b);
```

lexicographical_compare

Algorithmus der STL, definiert in `algorithm`.

Vergleicht die Objekte zweier Bereiche lexikografisch.

Varianten:

```
template<class Input1, class Input2>
bool lexicographical_compare(Input1 anf1, Input1 end1,
                             Input2 anf2, Input2 end2);

template<class Input1, class Input2, class BinPred>
bool lexicographical_compare(Input1 anf1, Input1 end1,
                  Input2 anf2, Input2 end2, BinPred fkt);
```

log

Funktion der Standardbibliothek, definiert in `cmath`.

Logarithmus Naturalis: berechnet den natürlichen Logarithmus von `a` zur Basis e (ln a oder $\log_{\varepsilon}$ a).

```
double log(double a);
```

log10

Funktion der Standardbibliothek, definiert in `cmath`.

Dekadischer Logarithmus: berechnet den Logarithmus von `a` zur Basis 10 ($\log_{10}$ a.)

```
double log10(double a);
```

long

Ganzzahliger, vorzeichenbehafteter, elementarer Datentyp. Mindestens 4 Bytes groß.

long double

Mit Vorzeichen behafteter, elementarer Fließkomma-Datentyp. Mindestens so groß wie `double`, im Normalfall aber größer. Die genaue Größe ist implementierungsabhängig.

lower_bound

Algorithmus der STL, definiert in `algorithm`.

Sucht in einem entsprechenden Bereich nach dem ersten Bereich, dessen Objekte gleich mit einem Objekt sind, und liefert die Startposition des gefundenen Bereichs zurück.

Varianten:

```
template<class Forward, class Typ>
Forward lower_bound(Forward anf, Forward end,
                    const Typ& obj);

template<class Forward, class Typ, class BinPred>
Forward lower_bound(Forward anf, Forward end,
                    const Typ &obj, BinPred fkt);
```

make_heap

Algorithmus der STL, definiert in `algorithm`.

Wandelt den entsprechenden Bereich in einen Heap um.

Varianten:

```
template<class Random>
void make_heap(Random anf, Random end);

template<class Random, class BinPred>
void make_heap(Random anf, Random end, BinPred fkt);
```

max

Algorithmus der STL, definiert in `algorithm`.

Liefert das größere der beiden Objekte.

Varianten:

```
template<class Typ>
const Typ &max(const Typ &o1, const Typ &o2);

template<class Typ, class BinPred>
const Typ &max(const Typ &o1, const Typ &o2, BinPred fkt);
```

max_element

Algorithmus der STL, definiert in `algorithm`.

Liefert das größte Objekt eines Bereichs.

Varianten:

```
template<class Forward>
Forward max_element(Forward anf, Forward end);

template<class Forward, class BinPred>
Forward max_element(Forward anf, Forward end, BinPred fkt);
```

memchr

Funktion der Standardbibliothek, definiert in `cstring`.

Sucht in den ersten `anz`-Zeichen des ab Adresse `adr` gelegenen Speicherblocks nach dem ersten Vorkommen des Zeichens `z`. Die Funktion liefert die Adresse des Zeichens oder `NULL` zurück.

```
void *memchr(const void *adr, int z, size_t anz);
```

memcmp

Funktion der Standardbibliothek, definiert in `cstring`.

Vergleicht die ersten `anz`-Zeichen der Speicherbereiche ab `adr1` und `adr2`.

Der Rückgabewert ist:

- negativ, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `adr1` kleiner ist als das von `adr2`.
- 0, wenn beide Speicherbereiche gleich sind.
- positiv, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `adr1` größer ist als das von `adr2`.

```
int memcmp(const void *adr1, const void *adr2, size_t anz);
```

memcpy

Funktion der Standardbibliothek, definiert in `cstring`.

Kopiert die ersten `anz`-Zeichen aus Speicherbereich `quelle` in den Speicherbereich `ziel`. Rückgabewert ist `ziel`.

Die Speicherbereiche dürfen sich nicht überlappen.

```
void *memcpy(const void *ziel, const void *quelle, size_t anz);
```

memmove

Funktion der Standardbibliothek, definiert in `cstring`.

Kopiert die ersten `anz`-Zeichen aus Speicherbereich `quelle` in den Speicherbereich `ziel`. Rückgabewert ist `ziel`.

Die Speicherbereiche dürfen sich überlappen.

```
void *memmove(const void *ziel, const void *quelle, size_t anz);
```

memset

Funktion der Standardbibliothek, definiert in `cstring`.

Beschreibt die ersten `anz`-Zeichen des ab Adresse `adr` gelegenen Speicherblocks mit dem Zeichen `z`. Rückgabewert ist `adr`.

```
void *memset(const void *adr, int z, size_t anz);
```

merge

Algorithmus der STL, definiert in `algorithm`.

Verschmilzt den ersten sortierten Bereich mit dem zweiten sortierten Bereich zu einem sortierten Bereich, der ab `anf3` abgespeichert wird.

Varianten:

```
template<class Input1, class Input2, class Output>
Output merge(Input1 anf1, Input1 end1,
             Input2 anf2, Input2 end2, Output anf3);

template<class Input1, class Input2, class Output,
         class BinPred>
Output merge(Input1 anf1, Input1 end1, Input2 anf2,
             Input2 end2, Output anf3, BinPred fkt);
```

min

Algorithmus der STL, definiert in `algorithm`.

Liefert das kleinere der beiden Objekte.

Varianten:

```
template<class Typ>
const Typ &min(const Typ &o1, const Typ &o2);

template<class Typ, class BinPred>
const Typ &min(const Typ &o1, const Typ &o2, BinPred fkt);
```

min_element

Algorithmus der STL, definiert in `algorithm`.

Liefert das kleinste Objekt eines Bereichs.

Varianten:

```
template<class Forward>
Forward min_element(Forward anf, Forward end);

template<class Forward, class BinPred>
Forward min_element(Forward anf, Forward end, BinPred fkt);
```

mismatch

Algorithmus der STL, definiert in `algorithm`.

Vergleicht die Objekte zweier Bereiche und liefert das erste ungleiche Paar zurück (oder die Ende-Positionen der beiden Bereiche).

Varianten:

```
template<class Input1, class Input2>
pair<Input1, Input2> mismatch(Input1 anf1, Input1 end1,
                              Input2 anf2);

template<class Input1, class Input2, class BinPred>
pair<Input1, Input2> mismatch(Input1 anf1, Input1 end1,
                              Input2 anf2, BinPred fkt);
```

modf

Funktion der Standardbibliothek, definiert in `cmath`.

Die Fließkommazahl `a` wird in einen ganzzahligen Wert i und einen Fließkommawert f, der die Nachkommastellen enthält, aufgespalten, sodass a=i+f gilt. i wird in `b` gespeichert und f von der Funktion zurückgegeben. i und f haben beide das gleiche Vorzeichen wie `a`.

```
double modf(double a, int *b);
```

mutable

Mit diesem Schlüsselwort werden Attribute deklariert, die auch dann noch variabel sein sollen, wenn das enthaltene Objekt konstant ist.

namespace

Mit `namespace` können Namensbereiche definiert werden, um Namenskonflikte zu vermeiden.

new

Reserviert Speicher für einen bestimmten Datentyp und liefert die Adresse des Speichers zurück.

```
// Speicherreservierung für ein Objekt:
CFileInfo *fptr = new CFileInfo;
// Speicherreservierung für 10 Objekte:
CFileInfo *fptr2 = new CFileInfo[10];
// Freigabe der Speicherbereiche:
delete(fptr);
delete(fptr2);
```

next_permutation

Algorithmus der STL, definiert in `algorithm`.

Betrachtet den entsprechenden Bereich als Permutation und wandelt ihn in die lexikografisch nachfolgende Permutation um. Ist die aktuelle Permutation bereits die lexikografisch letzte, dann wird die lexikografisch erste Permutation erzeugt und `false` zurückgegeben. Andernfalls ist der Rückgabewert `true`.

Varianten:

```
template<class Bi>
bool next_permutation(Bi anf, Bi end);
template<class Bi, class BinPred>
bool next_permutation(Bi anf, Bi end, BinPred fkt);
```

nth_element

Algorithmus der STL, definiert in `algorithm`.

Ein Bereich wird so weit sortiert, dass an Position `pos` das richtige Element steht. Alle Objekte im Bereich vor `pos` sind kleiner und alle Objekte im Bereich hinter `pos` sind größer als das Objekt an der Stelle `pos`.

Varianten:

```
template<class Random>
void nth_element(Random anf, Random pos, Random end);

template<class Random, class BinPred>
void nth_element(Random anf, Random pos, Random end,
                 BinPred fkt);
```

NULL

Eine in verschiedenen Header-Dateien definierte Konstante. Meist als `0`, `0L` oder `void*(0)` definiert.

partial_sort

Algorithmus der STL, definiert in `algorithm`.

Ein Bereich wird so weit sortiert, dass im Bereich vor Position `pos` alle Objekte ihre korrekte Position besitzen.

Varianten:

```
template<class Random>
void partial_sort(Random anf, Random pos, Random end);

template<class Random, class BinPred>
void partial_sort(Random anf, Random pos, Random end,
                  BinPred fkt);
```

partial_sort_copy

Algorithmus der STL, definiert in `algorithm`.

Der zweite Bereich wird durch sortierte Elemente des ersten Bereichs überschrieben.

Varianten:

```
template<class Input, class Random>
```

```
Random partial_sort_copy(Input anf, Input end,
                         Random anf2, Random end2);

template<class Input class Random, class BinPred>
Random partial_sort_copy(Input anf, Input end,
                    Random anf2, Random end2, BinPred fkt);
```

partition

Algorithmus der STL, definiert in `algorithm`.

Ein Bereich wird so sortiert, dass alle Objekte, für die ein Funktionsaufruf einen wahren Wert ergibt, vor den Objekten stehen, für die der Funktionsaufruf einen falschen Wert ergibt.

Zurückgegeben wird die Iterator-Position des ersten Objektes, für den der Funktionsaufruf einen falschen Wert lieferte.

```
template<class Bi, class Pred>
Bi partition(Bi anf, Bi end, Pred fkt);
```

pop_heap

Algorithmus der STL, definiert in `algorithm`.

Verschiebt das erste Element des als Heap organisierten Bereichs an das Ende des Bereichs.

Varianten:

```
template<class Random>
void pop_heap(Random anf, Random end);

template<class Random, class BinPred>
void pop_heap(Random anf, Random end, BinPred fkt);
```

pow

Funktion der Standardbibliothek, definiert in `cmath`.

Potenzfunktion: berechnet den Wert von `a` zur `b`-ten Potenz (a^b.).

```
double pow(double a, double b);
```

prev_permutation

Algorithmus der STL, definiert in `algorithm`.

Betrachtet den entsprechenden Bereich als Permutation und wandelt ihn in die lexikografisch vorhergehende Permutation um. Ist die aktuelle Permutation bereits die lexikografisch erste, dann wird die lexikografisch letzte Permutation erzeugt und `false` zurückgegeben. Andernfalls ist der Rückgabewert `true`.

Varianten:

```
template<class Bi>
bool prev_permutation(Bi anf, Bi end);

template<class Bi, class BinPred>
bool prev_permutation(Bi anf, Bi end, BinPred fkt);
```

private

Ein in Klassen und Strukturen verwendetes Zugriffsrecht, das Methoden und Attribute deklariert, die nur von Methoden der eigenen Klasse/Struktur und Freunden der Klasse/Struktur angesprochen werden dürfen.

protected

Ein in Klassen und Strukturen verwendetes Zugriffsrecht, das Methoden und Attribute deklariert, die nur von Methoden der eigenen Klasse/Struktur, Freunden der Klasse/Struktur und von abgeleiteten Klassen/Strukturen als Bestandteil der Unterklasse angesprochen werden dürfen.

public

Ein in Klassen und Strukturen verwendetes Zugriffsrecht, das Methoden und Attribute deklariert, die ohne jegliche Beschränkung angesprochen werden dürfen.

push_heap

Algorithmus der STL, definiert in `algorithm`.

Ist der entsprechende Bereich abgesehen vom letzten Element ein gültiger Heap, so wird das letzte Element des Bereichs derart in den Heap integriert, dass die Heap-Struktur erhalten bleibt.

Varianten:

```
template<class Random>
void push_heap(Random anf, Random end);

template<class Random, class BinPred>
void push_heap(Random anf, Random end, BinPred fkt);
```

random_shuffle

Algorithmus der STL, definiert in `algorithm`.

Die Objekte des entsprechenden Bereichs werden gleichverteilt zufällig gemischt.

```
template<class Random>
void random_shuffle(Random anf, Random end);
```

Referenz

Referenzen sind die in C++ hinzugekommene Möglichkeit, einen Verweis auf ein Objekt zu definieren:

```
int x=5;
int &r=x;
cout << r << endl;
```

Obwohl Referenzen bei ihrer Definition initialisiert werden müssen und deswegen nur auf ein Objekt verweisen können, benötigen sie im Gegensatz zu den Zeigern beim späteren Gebrauch keine explizite Syntax.

Siehe auch: Zeiger.

remove

Algorithmus der STL, definiert in `algorithm`.

Löscht alle Objekte eines Bereichs, die mit einem bestimmten Objekt gleich sind[1]. Rückgabewert ist die Position hinter dem letzten, nicht gelöschten Objekt.

1. Die Objekte werden nicht tatsächlich gelöscht, sondern an das Ende des Bereichs verschoben.

```
template<class Forward, class Typ>
Forward remove(Forward anf, Forward end, const Typ &obj);
```

remove_copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen durch Ersetzen der Objekte. Alle mit `obj` gleichen Objekte des ersten Bereichs werden dabei nicht berücksichtigt.

```
template<class Input, class Output, class Typ>
Output remove_copy(Input anf1, Input end1,
                   Output anf2, const Typ &obj);
```

remove_copy_if

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen durch Ersetzen der Objekte. Alle Objekte des ersten Bereichs, für die ein Funktionsaufruf einen wahren Wert liefert, werden dabei nicht berücksichtigt.

```
template<class Input, class Output, class Pred>
Output remove_copy_if(Input anf1, Input end1,
                      Output anf2, Pred fkt);
```

remove_if

Algorithmus der STL, definiert in `algorithm`.

Löscht alle Objekte eines Bereichs, für die ein Funktionsaufruf einen wahren Wert liefert[2]. Rückgabewert ist die Position hinter dem letzten, nicht gelöschten Objekt.

```
template<class Forward, class Pred>
Forward remove_if(Forward anf, Forward end, Pred fkt);
```

2. Die Objekte werden nicht tatsächlich gelöscht, sondern an das Ende des Bereichs verschoben.

replace

Algorithmus der STL, definiert in `algorithm`.

Ersetzt in einem Bereich alle mit `awert` gleichen Objekte durch das Objekt `nwert`.

```
template<class Forward, class Typ>
void replace(Forward anf, Forward end,
             const Typ &awert, const Typ &nwert);
```

replace_copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden. Dabei werden alle mit `awert` gleichen Objekte durch das Objekt `nwert` ersetzt.

```
template<class Input, class Output, class Typ>
Output replace_copy(Input anf1, Input end1, Output anf2,
                    const Typ &awert, const Typ &nwert);
```

replace_copy_if

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden. Dabei werden alle Objekte, für die ein Funktionsaufruf einen wahren Wert liefert, durch das Objekt `nwert` ersetzt.

```
template<class Input, class Output, class Pred, class Typ>
Output replace_copy_if(Input anf1, Input end1, Output anf2,
                       Pred fkt, const Typ &nwert);
```

replace_if

Algorithmus der STL, definiert in `algorithm`.

Ersetzt alle Objekte eines Bereichs, für die ein Funktionsaufruf einen wahren Wert liefert, durch das Objekt `nwert`.

```
template<class Forward, class Pred, class Typ>
void replace_if(Forward anf, Forward end, Pred fkt,
                const Typ &nwert);
```

reinterpret_cast

Wandelt einen Ausdruck in einen anderen Typ um. Es findet keine Typüberprüfung statt.

```
reinterpret_cast<Typ>(Ausdruck)
```

return

Mit dieser Anweisung springt das Programm aus einer Methode/Funktion heraus zurück zum Funktionsaufruf. Wenn die Methode/Funktion einen Wert zurückliefert, so kann dieser hinter `return` spezifiziert werden.

reverse

Algorithmus der STL, definiert in `algorithm`.

Die Objekte des entsprechenden Bereichs werden in ihrer Reihenfolge umgedreht.

```
template<class Bi>
void reverse(Bi anf, Bi end);
```

reverse_copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden, und dreht dabei die Originalreihenfolge der Objekte um.

```
template<class Bi, class Output>
Output reverse_copy(Bi anf1, Bi end1, Output anf2);
```

rotate

Algorithmus der STL, definiert in `algorithm`.

Rotiert die Objektreihenfolge durch Verschieben des Bereichs derart, dass das vorher an Iterator-Position `mit` befindliche Objekt danach an Iterator-Position `anf` steht.

```
template<class Forward>
void rotate(Forward anf, Forward mit, Forward end);
```

rotate_copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen, indem bereits vorhandene Objekte ersetzt werden. Dabei wird die ursprüngliche Reihenfolge der Objekte durch Verschieben derart rotiert, dass das vorher an Iterator-Position `mit` befindliche Objekt im Zielbereich an Iterator-Position `anf` steht.

```
template<class Forward, class Output>
Output rotate_copy(Forward anf, Forward mit, Forward end,
                   Output anf2);
```

search

Algorithmus der STL, definiert in `algorithm`.

Sucht in der ersten Sequenz nach dem ersten Vorkommen der zweiten Sequenz.

Varianten:

```
template<class Forward1, class Forward2>
Forward1 search(Forward1 anf1, Forward1 end1,
                Forward2 anf2, Forward2 end2);

template<class Forward1, class Forward2, class BinPred>
Forward1 find_end(Forward1 anf1, Forward1 end1,
                  Forward2 anf2, Forward2 end2, BinPred fkt);
```

search_n

Algorithmus der STL, definiert in `algorithm`.

Sucht nach mehreren, hintereinander liegenden, gleichen Objekten.

Varianten:

```
template<class Forward1, class Anzahl, class Typ>
Forward1 search_n(Forward1 anf1, Forward1 end1,
                  Anzahl anz, const Typ &obj);

template<class Forward1, class Anzahl,
         class Typ, class BinPred>
Forward1 search_n(Forward1 anf1, Forward1 end1, Anzahl anz,
                  const Typ &obj, BinPred fkt);
```

set_difference

Algorithmus der STL, definiert in `algorithm`.

Bildet die Differenz der beiden Mengen und legt das Ergebnis ab Position `anf3` ab.

Varianten:

```
template<class Input1, class Input2, class Output>
Output set_difference(Input1 anf1, Input1 end1,
                      Input2 anf2, Input2 end2, Output anf3);

template<class Input1, class Input2, class Output,
         class BinPred>
Output set_difference(Input1 anf1, Input1 end1,
       Input2 anf2, Input2 end2, Output anf3, BinPred fkt);
```

set_intersection

Algorithmus der STL, definiert in `algorithm`.

Bildet die Schnittmenge der beiden Mengen und legt das Ergebnis ab Position `anf3` ab.

Varianten:

```
template<class Input1, class Input2, class Output>
Output set_intersection(Input1 anf1, Input1 end1,
                    Input2 anf2, Input2 end2, Output anf3);

template<class Input1, class Input2, class Output,
         class BinPred>
Output set_intersection(Input1 anf1, Input1 end1,
     Input2 anf2, Input2 end2, Output anf3, BinPred fkt);
```

set_symmetric_difference

Algorithmus der STL, definiert in `algorithm`.

Bildet die symmetrische Differenz der beiden Mengen und legt das Ergebnis ab Position `anf3` ab.

Varianten:

```
template<class Input1, class Input2, class Output>
Output set_symmetric_difference(Input1 anf1, Input1 end1,
                    Input2 anf2, Input2 end2, Output anf3);

template<class Input1, class Input2, class Output,
         class BinPred>
Output set_symmetric_difference(Input1 anf1, Input1 end1,
       Input2 anf2, Input2 end2, Output anf3, BinPred fkt);
```

set_union

Algorithmus der STL, definiert in `algorithm`.

Bildet die Vereinigungsmenge der beiden Mengen und legt das Ergebnis ab Position `anf3` ab.

Varianten:

```
template<class Input1, class Input2, class Output>
Output set_union(Input1 anf1, Input1 end1,
                 Input2 anf2, Input2 end2, Output anf3);
```

```
template<class Input1, class Input2, class Output,
         class BinPred>
Output set_union(Input1 anf1, Input1 end1, Input2 anf2,
                 Input2 end2, Output anf3, BinPred fkt);
```

short

Ganzzahliger, vorzeichenbehafteter, elementarer Datentyp. Mindestens 1 Byte groß.

signed

Schlüsselwort zur Deklaration eines vorzeichenbehafteten elementaren Datentyps. Nur notwendig, wenn Datentyp in seiner Grundform vorzeichenlos ist (wie z.B. `char`.)

sin

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnet den Sinus von `a` im Bogenmaß.

```
double sin(double a);
```

sinh

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnet den Sinus Hyperbolicus von `a` im Bogenmaß.

```
double sinh(double a);
```

size_t

Ein in verschiedenen Header-Dateien definierter vorzeichenloser und ganzzahliger Variablentyp (oft `unsigned int`).

sizeof

Mit dem `sizeof`-Operator kann die Größe eines Datentyps oder Objekts in Bytes ermittelt werden.

sort

Algorithmus der STL, definiert in `algorithm`.

Sortiert einen Bereich mit dem nicht stabilen Sortierverfahren Quicksort.

Varianten:

```
template<class Random>
void sort(Random anf, Random end);

template<class Random, class BinPred>
void sort(Random anf, Random end, BinPred fkt);
```

sort_heap

Algorithmus der STL, definiert in `algorithm`.

Ist der entsprechende Bereich ein gültiger Heap, so wird er in eine sortierte Sequenz umgewandelt.

Varianten:

```
template<class Random>
void sort_heap(Random anf, Random end);

template<class Random, class BinPred>
void sort_heap(Random anf, Random end, BinPred fkt);
```

sqrt

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnet die Quadratwurzel von `a`.

```
double sqrt(double a);
```

stable_partition

Algorithmus der STL, definiert in `algorithm`.

Ein Bereich wird so sortiert, dass alle Objekte, für die ein Funktionsaufruf einen wahren Wert ergibt, vor den Objekten stehen, für die der Funktionsaufruf einen falschen Wert ergibt. Dabei bleibt die Reihenfolge der Objekte innerhalb eines Bereichs erhalten (stabile Sortierung.)

Zurückgegeben wird die Iterator-Position des ersten Objektes, für den der Funktionsaufruf einen falschen Wert lieferte.

```
template<class Bi, class Pred>
Bi stable_partition(Bi anf, Bi end, Pred fkt);
```

stable_sort

Algorithmus der STL, definiert in `algorithm`.

Sortiert einen Bereich mit dem stabilen Sortierverfahren Mergesort.

Varianten:

```
template<class Random>
void stable_sort(Random anf, Random end);

template<class Random, class BinPred>
void stable_sort(Random anf, Random end, BinPred fkt);
```

static

Dieses Schlüsselwort hat viele Bedeutungen:

- Bei Attributen werden mit `static` Klassenattribute deklariert, Attribute, die nur einmal für alle Objekte der Klasse existieren.
- Bei Methoden bedeutet `static`, dass die Methode nicht über ein Objekt der Klasse aufgerufen werden muss, sondern direkt über den Klassennamen aufgerufen werden kann. Voraussetzung dafür ist, dass die statische Methode nicht auf Attribute der Klasse oder auf Methoden, die auf Attribute der Klasse ansprechen, zugreift.
- Als `static` deklarierte lokale Variablen werden nicht nach Verlassen ihres Bezugsrahmens gelöscht, sondern besitzen ihren alten Wert noch, wenn der Bezugsrahmen erneut betreten wird.

static_cast

Wandelt einen Ausdruck in einen anderen Typ um. Es findet eine Typüberprüfung zur Kompilierzeit statt.

```
static_cast<Typ>(Ausdruck)
```

strcat

Funktion der Standardbibliothek, definiert in `cstring`.

Hängt den C-String `s2` durch Kopieren an den C-String `s1` an. Rückgabewert ist `s1`.

```
char *strcat(char *s1, const char *s2);
```

strchr

Funktion der Standardbibliothek, definiert in `cstring`.

Sucht im C-String `s` nach dem ersten Vorkommen des Zeichens `z`. Entweder wird die Position des gefundenen Zeichens oder `NULL` zurückgeliefert.

```
char *strchr(const char *s, int z);
```

strcmp

Funktion der Standardbibliothek, definiert in `cstring`.

Vergleicht die beiden C-Strings `s1` und `s2`. Der Rückgabewert ist:

- negativ, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `s1` kleiner ist als das von `s2`.
- 0, wenn beide C-Strings gleich sind.
- positiv, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `s1` größer ist als das von `s2`.

```
int strcmp(const char *s1, const char *s2);
```

strcoll

Funktion der Standardbibliothek, definiert in `cstring`.

Funktionalität wie `strcmp`, nur dass beim Zeichenvergleich lokalisierte Vergleichsregeln angewendet werden.

```
int strcoll(const char *s1, const char *s2);
```

strcpy

Funktion der Standardbibliothek, definiert in `cstring`.

Kopiert den C-String `quelle` an die Position `ziel`. Rückgabewert ist `ziel`.

```
char *strcpy(char *ziel, const char *quelle);
```

strcspn

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt den Index des ersten Zeichens in `s1`, das ebenfalls in `s2` enthalten ist.

```
int strcspn(const char *s1, const char *s2);
```

strerror

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt die zur Fehler-Nummer `fehler` gehörende Fehlermeldung.

```
char *strerror(int fehler);
```

strlen

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt die Anzahl der Zeichen im String `s` ohne Ende-Kennung.

```
size_t strlen(const char *s);
```

strncat

Funktion der Standardbibliothek, definiert in `cstring`.

Hängt maximal `laenge` Zeichen des C-Strings `s2` durch Kopieren an den C-String `s1` an. Bei Bedarf wird eine Ende-Kennung angehängt. Rückgabewert ist `s1`.

```
char *strncat(char *s1, const char *s2, size_t laenge);
```

strncmp

Funktion der Standardbibliothek, definiert in `cstring`.

Vergleicht maximal die ersten `laenge` Zeichen der C-Strings `s1` und `s2`. Der Rückgabewert ist:

- negativ, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `s1` kleiner ist als das von `s2`.
- 0, wenn beide C-Strings gleich sind.
- positiv, wenn beim ersten unterschiedlichen Zeichen das Zeichen von `s1` größer ist als das von `s2`.

```
int strncmp(const char *s1, const char *s2, size_t laenge);
```

strncpy

Funktion der Standardbibliothek, definiert in `cstring`.

Kopiert maximal `laenge` Zeichen des C-Strings `quelle` an die Position `ziel`. Bei Bedarf wird eine Ende-Kennung angehängt. Rückgabewert ist `ziel`.

```
char *strncpy(char *ziel, const char *quelle, size_t laenge);
```

strpbrk

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt die Adresse des ersten Zeichens in `s1` aus `s2`. Die Funktion liefert `NULL` zurück, wenn kein Zeichen gefunden wurde.

```
char *strpbrk(const char *s1, const char *s2);
```

strrchr

Funktion der Standardbibliothek, definiert in `cstring`.

Sucht im C-String `s` nach dem letzten Vorkommen des Zeichens `z`. Entweder wird die Position des gefundenen Zeichens oder `NULL` zurückgeliefert.

```
char *strrchr(const char *s, int z);
```

strspn

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt den Index des ersten Zeichens in `s1`, das nicht in `s2` enthalten ist.

```
int strspn(const char *s1, const char *s2);
```

strstr

Funktion der Standardbibliothek, definiert in `cstring`.

Ermittelt die Adresse des ersten Vorkommens von `s2` in `s1`. Die Funktion liefert `NULL` zurück, wenn die Suche erfolglos war.

```
char *strstr(const char *s1, const char *s2);
```

strtod

Funktion der Standardbibliothek, definiert in `cstdlib`.

Liefert einen `double`-Wert, der dem Wert entspricht, den die Zeichen des Strings `s` darstellen. Die in `s` enthaltene Zahl kann jedes von C++ bei der Ausgabe von `double`-Werten erlaubte Format besitzen.

Ist `ende` ungleich 0, wird in `*ende` die Adresse des Zeichens, das hinter dem letzten interpretierten Zeichen steht, gespeichert. Konnte keine Umwandlung erfolgen, ist `*ende` gleich `s`.

```
double strtod(const char *s, char **ende);
```

strtol

Funktion der Standardbibliothek, definiert in `cstdlib`.

Liefert einen `long`-Wert, der dem Wert entspricht, den die Zeichen des Strings `s` darstellen. Die in `s` enthaltene Zahl kann jedes von C++ bei der Ausgabe von `long`-Werten erlaubte Format besitzen.

Ist `ende` ungleich 0, wird in `*ende` die Adresse des Zeichens, das hinter dem letzten interpretierten Zeichen steht, gespeichert. Konnte keine Umwandlung erfolgen, ist `*ende` gleich `s`.

`basis` gibt das Zahlensystem der in `s` gespeicherten Zahl an. Ist `basis` gleich 0, wird die Erkennung des Zahlensystems anhand der in C++ üblichen Zahlendarstellung vorgenommen.

```
long strtol(const char *s, char **ende, int basis);
```

strtoul

Funktion der Standardbibliothek, definiert in `cstdlib`.

Funktionsweise wie `strtol`, nur dass ein `unsigned long`-Wert zurückgeliefert wird.

```
long strtoul(const char *s, char **ende, int basis);
```

struct

Schlüsselwort zur Definition einer Struktur.

swap

Algorithmus der STL, definiert in `algorithm`.

Vertauscht zwei Objekte.

```
template<class Typ>
inline void swap(Typ &o1, Typ &o2);
```

swap_ranges

Algorithmus der STL, definiert in `algorithm`.

Vertauscht zwei Bereiche.

```
template<class Forward1, class Forward2>
Forward2 swap_ranges(Forward1 anf1, Forward1 end1,
                     Forward2 anf2);
```

switch

Mit einem `switch`-Block können in Abhängigkeit vom Wert einer Variablen bestimmte Sprungmarken im Block angesprungen werden:

```
switch(x) {
  case 3: /* mach was */
  break;

  case 8: /* mach was */
  break;

  default: /* mach was */
  break;
}
```

Sollte `x` den Wert 3 oder 8 haben, so wird die dazugehörige Sprungmarke angesprungen. Sollte `x` einen Wert haben, für den keine spezielle Sprungmarke existiert, dann wird `default` angesprungen.

tan

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnet den Tangens von `a` im Bogenmaß.

```
double tan(double a);
```

tanh

Funktion der Standardbibliothek, definiert in `cmath`.

Berechnet den Tangens Hyperbolicus von `a` im Bogenmaß.

```
double tanh(double a);
```

template

Mit diesem Schlüsselwort kann eine Schablone mit variablen Datentypen definiert werden:

```
template<typename Typ>
Typ maximum(Typ a, Typ b) {
  return((>b)?a:b);
}
```

this

Jede nicht-statische Methode besitzt einen Zeiger namens `this`, der auf das Objekt verweist, über das die Methode aufgerufen wurde.

throw

Mit `throw` kann eine Ausnahme geworfen werden. Dazu wird hinter `throw` das Objekt angegeben, das geworfen werden soll.

tolower

Funktion der Standardbibliothek, definiert in `cctype`.

Sollte `a` ein Großbuchstabe sein, dann liefert `toupper` den dazugehörigen Kleinbuchstaben. Ist `a` kein Großbuchstabe, dann wird `a` selbst zurückgegeben.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int tolower(int a);
```

toupper

Funktion der Standardbibliothek, definiert in `cctype`.

Sollte `a` ein Kleinbuchstabe sein, dann liefert `toupper` den dazugehörigen Großbuchstaben. Ist `a` kein Kleinbuchstabe, dann wird `a` selbst zurückgegeben.

Für eine lokalisierte Anpassung der Funktion siehe Kapitel »Strings« Rezept 23.

```
int toupper(int a);
```

transform

Algorithmus der STL, definiert in `algorithm`.

Ruft für jedes Element des ersten Bereiches eine Funktion auf und speichert deren Rückgabewert im zweiten Bereich.

```
template<class Input, class Output, class Op>
Output transform(Input anf1, Input end1,
                 Output anf2, Op fkt);
```

Eine andere Variante betrachtet die Objekte des ersten und zweiten Bereichs als Paar und ruft eine Funktion mit diesem Paar auf. Die Ergebnisse der Funktionsaufrufe werden im dritten Bereich abgespeichert.

```
template<class Input1, class Input2,
         class Output, class BinOp>
Output transform(Input1 anf1, Input1 end1,
                 Input2 anf2, Output anf3, BinOp fkt);
```

true

Einer der Werte, die der boolesche Datentyp `bool` annehmen kann. `true` steht für »wahr« und besitzt normalerweise den numerischen Wert 1.

try

Wird innerhalb eines `try`-Anweisungsblocks eine Ausnahme geworfen, so kann sie mit hinter dem `try`-Block definierten `catch`-Blöcken aufgefangen werden.

typedef

Mit `typedef` können Datentypen neue Namen (Alias) gegeben werden.

```
typedef unsigned int UINT;
```

Der Variablentyp `unsigned int` kann nun auch über `UINT` benutzt werden.

typename

Spezifiziert den nachfolgenden Bezeichner als Typnamen. Kann auch bei Templates anstelle von `class` benutzt werden. Siehe auch: `template`.

unique

Algorithmus der STL, definiert in `algorithm`.

Löscht alle benachbarten Duplikate eines Bereichs[3]. Rückgabewert ist die Position hinter dem letzten, nicht gelöschten Objekt.

Varianten:

```
template<class Forward>
Forward unique(Forward anf, Forward end);

template<class Forward, class BinPred>
Forward unique(Forward anf, Forward end, BinPred fkt);
```

unique_copy

Algorithmus der STL, definiert in `algorithm`.

Kopiert einen Bereich in einen anderen durch Ersetzen der Objekte. Alle benachbarten Duplikate des ersten Bereichs werden dabei nicht berücksichtigt.

Varianten:

```
template<class Input, class Output>
Output unique_copy(Input anf1, Input end1, Output anf2);

template<class Input, class Output, class BinPred>
Output unique_copy(Input anf1, Input end1,
                   Output anf2, BinPred fkt);
```

using

Wenn Namen außerhalb ihres Namensbereichs angesprochen werden, so muss der Namensbereich immer mit angegeben werden:

```
std::cout << endl;
```

Mittels `using` kann deklariert werden, dass mit `cout` immer `std::cout` gemeint ist:

```
using std::cout;
```

3. Die Objekte werden nicht tatsächlich gelöscht, sondern an das Ende des Bereichs verschoben.

Auf diese Weise können auch ganze Namensbereiche global verfügbar gemacht werden:

```
using namespace std;
```

unsigned

Schlüsselwort zur Deklaration eines vorzeichenlosen elementaren Datentyps. Nur notwendig, wenn Datentyp in seiner Grundform vorzeichenbehaftet ist (wie z.B. `int`.) Nicht anwendbar auf Fließkomma-Datentypen.

unsigned int

Ganzzahliger, vorzeichenloser, elementarer Datentyp. Mindestens 2 Bytes groß.

unsigned long

Ganzzahliger, vorzeichenloser, elementarer Datentyp. Mindestens 4 Bytes groß.

unsigned short

Ganzzahliger, vorzeichenloser, elementarer Datentyp. Mindestens 1 Byte groß.

upper_bound

Algorithmus der STL, definiert in `algorithm`.

Sucht in einem entsprechenden Bereich nach dem ersten Bereich, dessen Objekte gleich mit einem Objekt sind, und liefert die Position hinter dem gefundenen Bereich zurück.

Varianten:

```
template<class Forward, class Typ>
Forward upper_bound(Forward anf, Forward end,
                    const Typ& obj);

template<class Forward, class Typ, class BinPred>
Forward upper_bound(Forward anf, Forward end,
                    const Typ &obj, BinPred fkt);
```

virtual

Deklariert eine Methode als virtuell. Wird eine virtuelle Methode über einen Zeiger oder eine Referenz aufgerufen, so wird zur Laufzeit geprüft, von welchem Datentyp das aufrufende Objekt wirklich ist.

void

Steht bei Funktionsparametern für »Kein Funktionsparameter« und bei Rückgabewerten für »Kein Rückgabewert«.

void wird auch zur Definition generischer Zeiger verwendet:

```
void *ptr=new int;
```

while

Ein Schleifenkonstrukt. Siehe Kapitel »Grundlagen« Rezept 6.

Zeiger

Zeiger bieten eine Möglichkeit, einen Verweis auf ein Objekt zu definieren. Dabei bedient man sich einer expliziten Syntax (Adressoperator, Dereferenzierung):

```
int x=5;
int *ptr;
ptr=&x;
cout << *ptr << endl;
```

Der wesentliche Vorteil von Zeigern gegenüber Referenzen liegt in der Möglichkeit, den Verweis nachträglich ändern zu können:

```
int a=5; b=10;
int *p;
p=&a;
cout << *p << endl;
p=&b;
cout << *p << endl;
```

Siehe auch: Referenz.

Stichwortverzeichnis

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt.
Dieses eBook stellen wir lediglich als **Einzelplatz-Lizenz** zur Verfügung!
Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschliesslich der Reproduktion, der Weitergabe, des Weitervertriebs, der Platzierung im Internet, in Intranets, in Extranets anderen Websites, der Veränderung, des Weiterverkaufs und der Veröffentlichung bedarf der schriftlichen Genehmigung des Verlags.
Bei Fragen zu diesem Thema wenden Sie sich bitte an:
mailto:info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.

Hinweis

Dieses und andere eBooks können Sie rund um die Uhr und legal auf unserer Website

(http://www.informit.de)

herunterladen