# 12

# Reguläre Ausdrücke

Dieses Kapitel behandelt die folgenden Themen:

- Grundlagen regulärer Ausdrücke
- Interaktiver Test
- Auszug der Programmierschnittstelle
- Praktische Anwendungen in C++

Ein regulärer Ausdruck ist eine Zeichenkette, die eine oder mehr grammatische Regeln enthält und zum Suchen oder zur Textersetzung dienen kann. Aus der Informatik wissen Sie, dass reguläre Ausdrücke mit einem endlichen Zustandsautomat (englisch *finite state machine*) verarbeitet werden können. Die Leistungsfähigkeit der Programmiersprache Perl basiert wesentlich auf regulären Ausdrücken. Die Programmiersprache Java stellt seit langem reguläre Ausdrücke zur Verfügung. So verwundert es nicht, dass das C++-Standardkomitee sich entschlossen hat, sie in den C++-Standard aufzunehmen.

Reguläre Ausdrücke sind ein leistungsfähiges Werkzeug. So lassen sich mit einem Befehl alle versehentlichen (und beabsichtigten) Wortverdopplungen wie »und und« in allen tex-Dateien eines Verzeichnisses auffinden, wobei die Klein- bzw. Großschreibung keine Rolle spielt:

```
egrep -n -i '\b([a-z]+) +\1\b' *.tex
```

Das folgende Kommando ersetzt jedes Vorkommen von »Version 1.2« und »Version 1.3« in der Datei *a.txt* durch »Version 2.0« bzw. »Version 3.0« und schreibt das Ergebnis in die Datei *b.txt*. Überzählige Leerzeichen nach »Version« werden gelöscht:

```
sed "s/ Version \+1\.\([23]\)/ Version \1.0/g" a.txt > b.txt
```

So wird aus »Nach Version 1.1 kamen Version  1.2 und Version 1.3.«:
»Nach Version 1.1 kamen Version 2.0 und Version 3.0.«

Warum das so ist, wird weiter unten erklärt. Dieses Kapitel konzentriert sich nach einer kurzen Einführung, die die wichtigsten Aspekte abdeckt, auf die Anwendung regulärer Ausdrücke mit C++. Zum tieferen Verständnis des Themas empfehle ich das exzellente Buch von Friedl [Fri] und als C++-spezifische Ergänzung [BeckP]. Es gibt mehrere Grammatiken (Dialekte) für reguläre Ausdrücke, die sich in großen Teilen nur wenig unterscheiden. Die gewünschte Grammatik kann im Konstruktor eines regulären Ausdrucks angegeben werden. So wird mit

```
regex rgx(regAusdruck, regex_constants::egrep);
```

die Grammatik gewählt, die von dem am Anfang des Kapitels erwähnten Programm *egrep* benutzt wird (`regAusdruck` ist eine Zeichenkette des Typs `const char*` oder `string`). Falls keine Grammatik angegeben wird, ist Ecma-262 [Ecma] mit einigen Modifikationen voreingestellt ([ISOC++], Kap. 28.13). Ecma-262, früher JavaScript, bezieht sich bezüglich der regulären Ausdrücke auf Perl 5. Im Folgenden wird von dieser Voreinstellung ausgegangen. Auch werden reguläre Ausdrücke auf ASCII-Zeichen des Typs `char` beschränkt, obwohl es Erweiterungen für `wchar_t`-Zeichen und Unicode gibt.

## 12.1 Elemente regulärer Ausdrücke

Wie an den einführenden Beispielen zu sehen, haben manche Zeichen besondere Bedeutung. Sie werden Meta-Zeichen genannt:

`.` Ein Punkt steht für ein beliebiges Zeichen. Der reguläre Ausdruck `.` trifft damit auf jedes Zeichen der Folge `abc123` zu.

`|` gibt eine Alternative an. `a|b` heißt `a` oder `b`.

`()` Runde Klammern dienen zur Gruppierung. Wenn `a`, gefolgt von `b` oder `c`, gemeint ist, schreibt man `a(b|c)`. Eine weitere Bedeutung ist die Gruppierung zwecks späterer Identifikation der Gruppe (Beispiel folgt).

`(?:)` Mit `(?:)` wird eine Gruppierung markiert, die *nicht* für eine späterer Identifikation vorgesehen ist. Beispiel: `(?:a)`.

`\` Der Backslash (Escape-Zeichen) hat mehrere Bedeutungen:

- Wenn er einem Meta-Zeichen vorangestellt wird, ist nicht das Meta-Zeichen gemeint, sondern das zugehörige Literal. Der reguläre Ausdruck »\.« trifft damit nur auf den Punkt in der Folge `ab.c123` zu.

- Ein \, gefolgt von einem besonderen Buchstaben, kann zu einem ASCII-Steuerzeichen oder einem Meta-Zeichen werden. So meint ^I das Tabulatorzeichen, während \d für eine beliebige Ziffer steht. Falls dem nachfolgenden Buchstaben keine besondere Bedeutung zukommt, ist dieser Buchstabe gemeint. So entspricht \m genau dem Buchstaben m (siehe Tabelle 12.1).

**Tabelle 12.1:** Escape-Zeichen

| Zeichen | Bedeutung |
|---|---|
| | Steuerzeichen (Auswahl): |
| \\ | Backslash |
| \f | Seitenvorschub |
| \n | neue Zeile |
| \r | CR |
| \t | horizontaler Tabulator |
| \v | vertikaler Tabulator |
| | Zeichenklassen: |
| \d | Ziffer, dasselbe wie [0-9] |
| \D | keine Ziffer, dasselbe wie [^0-9] oder [^\d] |
| \s | Zwischenraumzeichen, dasselbe wie [ \f\n\r\t\v] |
| \S | kein Zwischenraumzeichen, dasselbe wie [^\s] |
| \w | Wortzeichen, in ASCII [a-zA-Z0-9] |
| \W | kein Wortzeichen, d.h. [^\w] |
| | Begrenzer: |
| \b | Wortgrenze (siehe unten) |
| \B | keine Wortgrenze |

- Ein \, gefolgt von einer Ziffer, ist ein Verweis auf eine Gruppe, die vorher gefunden wurde. (abc)xy\1 stimmt daher mit abcxyabc überein. Nach xy steht \1 für die erste gefundene Gruppe, hier abc.
- Wenn ein Backslash-Literal gemeint ist, ist \\ zu schreiben. Da in einem C-String \ als \\ geschrieben wird, heißt das, dass ein Backslash, der als regulärer Ausdruck \\ geschrieben wird, in einem C-String vierfach, also als \\\\, auftritt.

`[]` kennzeichnet eine Zeichenklasse. [xy] passt auf alle x und alle y einer Folge. [a-z] passt auf jeden Kleinbuchstaben einer Zeichenfolge, [0-9] auf jede Ziffer.

`[^]` negiert die Bedeutung. [^a-z] passt auf jedes Zeichen, das kein Kleinbuchstabe ist.

`+` ist wie die folgenden Meta-Zeichen ein Quantifizierer. + passt nur, wenn das vorstehende Element mindestens einmal vorkommt. So passt a+ sowohl auf abc wie auf aaabc, nicht aber auf bc.

`?` passt, wenn das vorstehende Element einmal oder gar nicht vorkommt.

`*` passt, wenn das vorstehende Element beliebig oft vorkommt. Das schließt die Möglichkeit, dass das Element nicht vorkommt, mit ein.

`{n}` passt, wenn das vorstehende Element genau n-mal vorkommt. a{3} passt auf aaa, aber nicht auf aa. Die Erweiterung {n, m} passt, wenn das vorstehende Element mindestens n-mal und maximal m-mal vorkommt.

`^` Außerhalb einer Zeichenklasse ist `^` wie das folgenden Meta-Zeichen `$` ein *Anker*. Ein Anker markiert kein Zeichen, sondern eine Position. `^` passt auf den Anfang einer Zeile.

`$` passt auf das Ende einer Zeile oder eines Strings. Der reguläre Ausdruck `\d\d` passt auf die Teilfolgen 12 und 45 im String a123b456. Wird das `$`-Zeichen angefügt, d.h. `\d\d$`, passt der Ausdruck nur auf 56.

`\b` ist eine Wortgrenze. `a\b` meint also das Zeichen `a` am Ende eines Worts. `a\b` passt nirgends in `xyz abc`, wohl aber auf das erste `a` von `xyza abc`.

`\B` ist keine Wortgrenze. `a\B` passt nur auf das zweite `a` von `xyza abc`.

Die Zeichenklasse `[a-z]` bezeichnet die ASCII-Kleinbuchstaben. Die aus dem POSIX-Standard übernommene Zeichenklasse `[[:lower:]]` bezeichnet die Kleinbuchstaben in Abhängigkeit von der Locale-Einstellung, sodass bei Einstellung der Locale `de_DE` auch Umlaute erfasst werden. In Analogie dazu gibt es weitere Zeichenklassen, deren Namen sich aus den Funktionsnamen der Tabelle 35.2 auf Seite 875 ergeben, wenn das Präfix `is` weggelassen wird. Aus `islower()` ergibt sich `[[:lower:]]`.

Damit sind die wichtigsten Elemente genannt. Es gibt weitere, die aus Platzgründen hier nicht aufgeführt werden. Mit der Bitte um Verständnis verweise ich auf die schon erwähnte Literatur [Fri] und [BeckP].

#### Verknüpfungen

In Tabelle 12.1 ist in der Zeile zu `\w` zu sehen, dass in Zeichenklassen kombiniert werden kann. `[a-zA-Z0-9]` enthält die alphanumerischen Zeichen. Mit der `&&`-Verknüpfung lässt sich eine Subtraktion darstellen: `[a-z&&[^aeiouy]]` bedeutet, dass die Menge der Vokale von der Menge der Kleinbuchstaben abgezogen wird. Die Zeichenklasse enthält nur noch kleingeschriebene Konsonanten.

### 12.1.1 Greedy oder lazy?

Bei der Auswertung regulärer Ausdrücke wird versucht, eine passende Zeichenkette möglichst großer Länge zu finden. Deshalb wird dieses Vorgehen »gierig« (englisch *greedy*) genannt. Die in der obigen Aufstellung genannten Quantifizierer `+`, `*`, `?` usw. sind alle *greedy*. Manchmal möchte man aber die kürzestmögliche passende Zeichenkette finden. Dafür gibt es die »träge« oder »faule« (englisch *lazy*) Auswertung. *Lazy* Quantifizierer unterscheiden sich syntaktisch von Greedy-Quantifizierern durch ein nachgestelltes `?`, zum Beispiel `a??` für ein optionales `a`. Um den Unterschied zu verdeutlichen, wird ein regulärer Ausdruck gesucht, der aus einer Zeile einen Kommentar herausfiltern soll. Als Teststring wird

```
xyz /* hallo */ abc */ 123
```

vorgegeben. Sie sehen, dass es zwei Positionen für das Kommentarende `*/` gibt. Im C++-Sinn ist nur die erste korrekt.

#### Greedy ...

Betrachten Sie den regulären Ausdruck `/\*.*\*/`. Er passt auf alle Zeichenketten, die

- mit einem Schrägstrich / beginnen, gefolgt

- von einem *. Bitte beachten Sie, dass das Meta-Zeichen * mit einem Escape-Zeichen maskiert werden muss, weil das Literal und nicht der Quantifizierer gemeint ist.
- Anschließend folgen beliebige Zeichen, mit dem Meta-Zeichen . symbolisiert. Die Anzahl dieser Zeichen kann null oder beliebig groß sein, wie das nachfolgende Meta-Zeichen * festlegt.
- Abschließend folgt das Literal *, gefolgt von /.

Nun gehören auch die Zeichen * und / zu den beliebigen Zeichen, sodass die Auswertungsmaschinerie, die den ganzen String untersucht, weitermacht, bis das Ende erreicht ist. Die letzte unterwegs gefundene Möglichkeit, eine Übereinstimmung zu erreichen, ist das zweite Vorkommen von */. Der reguläre Ausdruck /*.**/ passt daher auf die Zeichenkette `/* hallo */ abc */` – nicht der gesuchte C++-Kommentar. Die »gierige« Strategie führt manchmal nicht zum Erfolg.

### ... oder lazy?

Die gierige Auswertung kann mit einem ?, das dem Quantifizierer folgt, verhindert werden. In diesem Fall lautet der modifizierte reguläre Ausdruck /*.*?*/. Angewendet auf den Teststring passt dieser Ausdruck auf die Zeichenkette `/* hallo */`, das heißt, es wird die erste passende Möglichkeit genommen.

**Tipp**

Vermeiden Sie die greedy-Suche mit .*, wenn Sie keinen besonderen Grund dafür haben.

Begründungen: .* ohne Lazy-Quantifizierer verschwendet Rechenzeit durch überflüssiges Durchlaufen der Zeichenkette und Backtracking. Das Beispiel `/* hallo */ ... 1000 Zeichen ohne Kommentarende ...` macht es deutlich. Außerdem ist manchmal die kürzeste passende Zeichenkette gewünscht, nicht die längste.

**Tipp**

Eine Alternative zu .* ist die Greedy-Suche mit einer Zeichenklasse, die die Negation eines zu suchenden Zeichens enthält.

Der Ausdruck "<.*>" findet HTML-Tags, sucht aber bis zum Ende. Die Alternative ist hier, den Punkt durch ein Zeichen, das nicht der Endekennung entspricht, zu ersetzen: "<[^>]*>". Diese greedy-Suche ist effizient, weil sie beim ersten > anhält.

## 12.2 Interaktive Auswertung

Reguläre Ausdrücke sind oft nicht leicht zu lesen. Die Änderung nur eines einzigen Zeichens verleiht dem Ausdruck eine andere Bedeutung, wie am obigen Beispiel zu sehen ist. Deswegen ist es sinnvoll, einen Ausdruck vor dem festen Einbau in ein Programm zu evaluieren. Die Klasse `RegexTester` ist dafür gut geeignet.

### Hinweis

Der Compiler g++ bis einschließlich Version 4.5 kann noch keine regulären Ausdrücke, weswegen Boost eingesetzt wird. Die regulären Ausdrücke der Boost-Bibliothek sind in den C++-Standard aufgenommen worden, nur sind die Compiler großenteils noch nicht angepasst.



**Listing 12.1:** Klasse RegexTester

```
// cppbuch/k12/regextester/RegexTester.h
#ifndef REGEX_TESTER
#define REGEX_TESTER
#include<boost/regex.hpp>       // siehe Hinweis oben
#include<string>

class RegexTester {
 public:
   RegexTester(const char* regEx, const char* teststr);
   void run();
 private:
   boost::regex rgx;            // siehe Hinweis oben
   std::string teststring;
};
#endif
```

Dem Hauptprogramm werden bei Aufruf ein regulärer Ausdruck und der auszuwertende String übergeben. Ausgegeben werden alle Zeichen und Positionen, auf die der reguläre Ausdruck zutrifft.

**Listing 12.2:** Hauptprogramm zum interaktiven Testen

```
// Auszug aus cppbuch/k12/regextester/testRegex.cpp
#include <iostream>
#include "RegexTester.h"
using namespace std;

int main(int argc, char* argv[]) {
   if(3 != argc) {
      cout << "Gebrauch: testRegex.exe \"regex\" \"teststring\"" << endl;
   }
   else {
      try {
         RegexTester rt(argv[1], argv[2]);
         rt.run();
      } catch(boost::regex_error& re) {
         std::cerr << "Fehler: " << re.what() << std::endl;
      }
   }
}
```

Die Implementation der Klasse `RegexTester` demonstriert bereits einige Möglichkeiten der Verarbeitung regulärer Ausdrücke mit C++. Das Listing wird unten im Detail diskutiert.

**Listing 12.3:** RegexTester-Implementierung

```
// cppbuch/k12/regextester/RegexTester.cpp
#include <iostream>
#include "RegexTester.h"

RegexTester::RegexTester(const char* regEx, const char* teststr)
     : rgx(regEx), teststring(teststr) {
}

void RegexTester::run() {
   boost::sregex_iterator erster(teststring.begin(), teststring.end(), rgx),
      letzter;
   std::cout << "Regex: " << rgx
            << "  Teststring: " << teststring << std::endl;
   if (erster == letzter) {
      std::cout << "nichts gefunden" << std::endl;
   }
   else {
      while(erster != letzter) {
         boost::match_results<std::string::const_iterator>
            ergebnis = *erster++;
         for(size_t i = 0; i < ergebnis.size(); ++i) {
            if(i > 0) {
               std::cout << "Capturing Group " << i << ": ";
            }
            std::cout << "\"" << ergebnis.str(i) << "\" gefunden. Position "
                     << ergebnis.position(i);
            if(ergebnis.length(i) > 1) {
               std::cout << " bis "
                        << ergebnis.position(i) + ergebnis.length(i)-1;
            }
            std::cout << std::endl;
         }
      }
   }
}
```

Der Konstruktor initialisiert das Attribut des Typs `regex` mit dem als C-String übergebenen regulären Ausdruck und merkt sich den Teststring. Die Klasse `regex` ist die Klasse für die Verarbeitung regulärer Ausdrücke.

Die Klasse `sregex_iterator` ist die Spezialisierung `regex_iterator<string::const_iterator>` der zugrunde liegenden Template-Klasse. Dem damit erzeugten Iterator `erster` werden der zu untersuchende Bereich des Teststrings und der zu verwendende reguläre Ausdruck übergeben. Der Iterator `letzter` hat die in der STL übliche Funktion, als Endekriterium zu dienen.

Die Klasse `match_results` ist eine Sammlung von Zeichenfolgen, die das Ergebnis der Auswertung repräsentieren. Es kann mehrere Ergebnisse geben, die der Iterator jeweils der Reihe nach dem Objekt `ergebnis` zuweist.

Der Aufruf von `ergebnis.size()` gibt 1 zurück plus die Anzahl gefundener Unterausdrücke, die eine Gruppe darstellen (englisch *capturing group*). In der Schleife werden das

(mit `str()` in einen String umgewandelte) Ergebnis und die gefundene Position ausgegeben. Wenn die Übereinstimmung mehrere Zeichen umfasst, wird auch die Endposition angezeigt. Die Beispiele zeigen den Programmaufruf und direkt danach die Ausgabe des Programms.

- Programmaufruf: `testRegex.exe "[a-c]+" "zcbaaxcadey"`

```
Regex: [a-c]+ Teststring: zcbaaxcadey
"cbaa" gefunden. Position 1 bis 4
"ca" gefunden. Position 6 bis 7
```

- Programmaufruf: `testRegex.exe "[a-c]*" "z"`

```
Regex: [a-c]* Teststring: z
"" gefunden. Position 0
"" gefunden. Position 1
```

  Die Erklärung: Der Quantifizierer `*` erlaubt es, dass ein Element gar nicht auftritt. Das ist hier am Anfang und am Ende gegeben.

- Programmaufruf: `testRegex.exe "^\\D+([0-9]+) \\1" "Sommer 2012 2012"`

```
Regex: ^\D+([0-9]+) \1 Teststring: Sommer 2012 2012
"Sommer 2012 2012" gefunden. Position 0 bis 15
Capturing Group 1: "2012" gefunden. Position 7 bis 10
```

  Erklärung: Wie oben schon erläutert, wird ein Backslash in einem C-String als `\\` dargestellt. Deswegen ist die Dopplung auch bei der Eingabe erforderlich. Dieser reguläre Ausdruck verlangt eine Folge von Nicht-Zifferzeichen (`\D`), die aus mindestens einem Zeichen besteht (`+`), und die am Anfang des Teststrings beginnt (`^`). Dies wird durch die Zeichenfolge »Sommer « einschließlich des Leerzeichens erfüllt. Es schließt sich eine Ziffernfolge (`[0-9]`) mit mindestens einer Ziffer an (`+`). Diese Ziffernfolge wird wegen der runden Klammern als Gruppe gespeichert. Nach den Ziffern folgt ein Leerzeichen und dann die erste gefundene Gruppe (`\1`), hier 2012.

## 12.3 Auszug des regex-APIs

Aus Platzgründen kann hier nur der wichtigste Teil der Programmierschnittstelle für reguläre Ausdrücke dargestellt werden. Weitere Informationen bitte ich, [ISOC++] zu entnehmen. Der Header ist `<regex>`. Die Klasse `regex` ist ein anderer Name für die Spezialisierung `basic_regex<char>`. Die entsprechende Klasse für `wchar_t` heißt `wregex`. Im folgenden Text beschränke ich mich auf `regex` und lasse auch viele vordefinierte Parameter weg, weil die Standardeinstellung meistens genügt.

- `regex(const char*)` und `regex(const string&)`
  sind Konstruktoren, die ein `regex`-Objekt nach dem vorgegebenen Standard Ecma-262 [Ecma] erzeugen. Falls ein anderer Standard gewünscht ist, kann er als Parameter übergeben werden, zum Beispiel

```
regex rgx(regAusdruck, regex_constants::egrep);
```

Außer `ECMAScript` und `egrep` sind auch `awk` und `grep` möglich. Die Konstante `icase` sorgt dafür, dass Groß- und Kleinschreibung nicht unterschieden werden.

- `sregex_iterator` ist ein Iterator, mit dem über die Ergebnisse der Auswertung eines regulären Ausdrucks iteriert werden kann. Dabei ist die auszuwertende Zeichenkette vom Typ `string`. Falls der Typ `const char*` ist, muss ein `cregex_iterator` genommen werden. Beide Iteratortypen sind Spezialisierungen des Klassen-Templates `regex_iterator`. Ein Beispiel finden Sie auf Seite 414.
- `match_results` ist das Klassen-Template zur Ergebnisablage. Ein `match_results`-Objekt ergibt sich aus der Dereferenzierung des `regex_iterators`. `match_results` hat unter anderem die Methoden `size()`, `position()` und `length()`. Das genannte Beispiel von Seite 414 zeigt die Verwendung von `match_results` und seiner Methoden.

### regex_match

- `bool regex_match(BidirectionalIterator first, BidirectionalIterator last, match_results& match, const regex& rgx)` stellt fest, ob der gesamte Bereich zwischen `first` und `last` dem regulären Ausdruck `rgx` entspricht. Falls ja, wird `true` zurückgegeben und `match` entsprechend geändert.
- `bool regex_match(const char* str, match_results& match, const regex& rgx)` gibt `regex_match(str, str+strlen(str), match, rgx)` zurück.
- `bool regex_match(const string& s, match_results& match, const regex& rgx)` gibt `regex_match(s.begin(), s.end(), match, rgx)` zurück.
- `bool regex_match(const char* str, const regex& rgx)` entspricht der zweitgenannten `regex_match()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.
- `bool regex_match(const string& s regex& rgx)` entspricht der an dritter Stelle genannten `regex_match()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.

### regex_search

- `bool regex_search(BidirectionalIterator first, BidirectionalIterator last, match_results& match, const regex& rgx)` durchsucht den Bereich zwischen `first` und `last` entsprechend dem regulären Ausdruck `rgx`. Falls er für eine Teilfolge dazwischen zutrifft, wird `true` zurückgegeben und `match` entsprechend geändert.
- `bool regex_search(const char* str, match_results& match, const regex& rgx)` gibt `regex_search(str, str+strlen(str), match, rgx)` zurück.
- `bool regex_search(const string& s, match_results& match, const regex& rgx)` gibt `regex_search(s.begin(), s.end(), match, rgx)` zurück.
- `bool regex_search(BidirectionalIterator first, BidirectionalIterator last, const regex& rgx)` entspricht der erstgenannten `regex_search()`-Funktion für den Fall, dass kein `match_results`-Objekt gebraucht wird.
- `bool regex_search(const char* str, const regex& rgx)` gibt `regex_search(str, str+strlen(str), rgx)` zurück.
- `bool regex_search(const string& s, const regex& rgx)` gibt `regex_search(s.begin(), s.end(), rgx)` zurück. Ein Beispiel finden Sie auf Seite 636.

### regex_replace

- `OutputIterator regex_replace(OutputIterator out, BidirectionalIterator first, BidirectionalIterator last, const regex& gesucht, string ersatz)`
  kopiert den Bereich zwischen `first` und `last` über den Iterator `out` in die Ausgabe, wobei mit `gesucht` gefundene Übereinstimmungen durch `ersatz` ersetzt werden.
- `string regex_replace(const string& alles, const regex& gesucht, const string& ersatz)` gibt einen String zurück, der `alles` entspricht, nur dass dabei mit `gesucht` gefundene Übereinstimmungen durch `ersatz` ersetzt werden. Ein Beispiel finden Sie auf Seite 638.

## 12.4 Anwendungen

Reguläre Ausdrücke können vielfältig eingesetzt werden. Um Dopplungen zu vermeiden, wird an dieser Stelle auf das Kapitel 24 »Algorithmen für verschiedene Aufgaben« verwiesen. Zu den Programmen dort gibt es weitergehende Erläuterungen. In Kapitel 24 werden Lösungen auf Basis regulärer Ausdrücke in den folgenden Abschnitten angeboten:

- Datei durchsuchen: Abschnitt 24.2.1, Seite 636.
  Verwendet werden `regex`, `regex::egrep`, `regex::icase`, `regex_search()`, `regex_error`.
- Ersetzungen in einer Datei vornehmen: Abschnitt 24.2.2, Seite 638.
  Verwendet werden `regex`, `regex::egrep`, `regex::icase`, `regex_replace()`, `regex_error`.
- Lines of Code ermitteln: Abschnitt 24.2.4, Seite 641.
  Verwendet werden `regex`, `regex_replace()`, `regex_error`.
- Erkennung eines Datums: Abschnitt 24.12.1, Seite 710.
  Verwendet werden `regex`, `regex_match()`.
- Erkennung einer IP-Adresse: Abschnitt 24.12.2, Seite 712.
  Verwendet werden `regex`, `regex_match()`.