
Looping Back To Start Of While Loop

Es ist unnütz, etwas mit mehr zu tun, was auch mit weniger getan werden kann.

Wilhelm von Ockham (1290 Ockham, Surrey – 1349 München),
das nach ihm benannte Rasiermesser beschreibend

Schlagwörter	Programmumformung, Programmtransformation, Restrukturierung, strukturiertes Programmieren, Unstetigkeit, Verständlichkeit, Codeblähung, Schleife, C++, <i>continue</i>
Motivation	Dieses Dokument ergänzt <i>Krimskrams – Algorithmen verbessern und messen</i> um ein Beispielprogramm, das restrukturiert und korrigiert wird.
Lernziele	Der Leser übt an einem Algorithmus, • ihn leserlich zu formatieren, • zu vereinfachen, • vermeidbare Unstetigkeiten zu entfernen, • unstrukturierte Teile in strukturierte Teile umzuformen, • Indexfehler und Pseudoschleifen zu erkennen und zu korrigieren.

Inhalt

1	Das Problem	1
2	Restrukturierungen	2
3	Korrekturen	10
4	Weitere Restrukturierungen	12
5	Die Lösung	13
A	Literatur	15
B	Aufgabenlösungen	15

1

Das Problem

Der Hilferuf

„Hi. I’m very new at C++. I need helping in this function to return to the top of the while loop. I’ve said in comments where this change needs to be done (within the inmost if statement). Any suggestions as to how I should go to the start of the while loop?

This function works how I want it to during the first iteration, but I can’t get it to return to the top.

Please help!

Thanks in advance!“ [DIC10]

von lakers1230 an Dream.In.Code, einer Online-Gemeinschaft für Programmierer und Webentwickler, zu seinem Programm 1, ist zu beantworten.

Programm 1C++: check original aus
[DIC10]

```

01 bool check(string way)
02 {
03     string copy = way;
04     while(copy.size() >= 0)
05     {
06         if(copy == "")
07             return true;
08         if(copy.size() == 1)
09             return false;
10         if(copy.at(0) == 'n' || copy.at(0) == 'N' || copy.at(0)
11 == 's' || copy.at(0) == 'S' || copy.at(0) == 'e' || copy.at(0) == 'E'
12 || copy.at(0) == 'w' || copy.at(0) == 'W')
13         {
14             if(isdigit(copy.at(1)))
15             {
16                 if(isdigit(copy.at(2)))
17                 {
18                     copy = copy.substr(0,2);
19                     //HERE
20                 }
21                 else
22                 {
23                     copy = copy.substr(0,1);
24                     //HERE
25                 }
26             }
27             return true;
28         }
29         else
30             return false;
31     }
32 }

```

2**Restrukturierungen**

Dass sich lakers1230 mit dem Ablauf seines Programms 1 identifiziert – „I should go to the start of the while loop“ –, sei als Flapsigkeit ignoriert. Seine Frage, wie sich der Ablauf vom Schleifenrumpf zur Bedingung der *while*-Schleife lenken lässt, ist schnell mit „Mit der *continue*-Anweisung!“ beantwortet. Die Stellen, an denen sich lakers1230 diesen Sprung wünscht, hat er kommentiert. Von Programm 1 zu Programm 2

Wunsch erfüllen

- ersetzen wir in den Zeilen 17 und 22 den Kommentar „//HERE“ durch „*continue*;“ und führen diese kosmetischen, lexikerhaltenden Änderungen durch:

Formatieren

- die für *Java* übliche platzfressende Festbreitschrift durch die von Stroustrup für C++ bevorzugte Proportionalschrift ersetzen,
- einige Leer- und Zeilenendezeichen ergänzen oder streichen.

Programm 2C++: check mit
continue

```

01  bool check (string way) {
03      string copy = way;
04      while (copy.size() >= 0) {
06          if (copy == "")
07              return true;
08          if (copy.size() == 1)
09              return false;
10          if (copy.at(0) == 'n' || copy.at(0) == 'N' || copy.at(0) == 's' || copy.at(0) == 'S' ||
              copy.at(0) == 'e' || copy.at(0) == 'E' || copy.at(0) == 'w' || copy.at(0) == 'W') {
12              if (isdigit(copy.at(1))) {
14                  if (isdigit(copy.at(2))) {
16                      copy = copy.substr(0, 2);
17                      continue;
18                  } else {
21                      copy = copy.substr(0, 1);
22                      continue;
23                  }
24                  return true;
25              } else
27                  return false;
28          } else
30              return false;
31      }
32  }

```

Wo lässt sich Programm 2 schnell vereinfachen, Ockhams Rasiermesser ansetzen? In Zeile 01 ist *way* ein Wertparameter der Funktion *check*, *way* erhält also eine Kopie des aktuellen Parameters als Wert. In Zeile 03 wird von *way* die Kopie *copy* angelegt. Danach wird nur noch auf *copy*, nicht mehr auf *way* zugegriffen. Da *way* änderbar ist und sein Anfangswert nach den Änderungen an *copy* in den Zeilen 16 und 21 nicht benötigt wird, können diese Änderungen an *way* statt an *copy* vorgenommen werden: *copy* ist überflüssig.

Unnütze Variable
entfernen

- Wir ersetzen alle Zugriffe auf *copy* durch Zugriffe auf *way* und streichen die Vereinbarung von *copy*.

In Zeile 04 ist die Bedingung *copy.size() >= 0* stets **true**, da es keine *string*-Objekte mit negativer Länge gibt. Der Indexbereich von C++-*string*-Objekten ist $0 \dots \text{size}() - 1$.

Ausdruck durch
äquivalente Konstante
ersetzen

- Wir ersetzen in Zeile 04 *copy.size() >= 0* durch **true**.

Scheinbar gehört der neue Schleifenkopf

```
04  while (true) {
```

zu einer Endlosschleife, aber da der Schleifenrumpf durch fünf **return**-Anweisungen verlassen wird, trügt der Schein: Es ist eine unstrukturierte Schleife mit fünf Ausgängen. Dies ist die größte Schwäche des Programms, aber der Hauptaufgabe, die Schleife in eine strukturierte Schleife umzuformen, widmen wir uns erst nach einigen einfacheren Restrukturierungen.

Bei der zweiseitigen Auswahlanweisung in den Zeilen 14 bis 23 enden beide Zweige mit einer **continue**-Anweisung:

```

14      if (isdigit(copy.at(2))) {
16          copy = copy.substr(0, 2);
17          continue;
18      } else {
21          copy = copy.substr(0, 1);
22          continue;
23      }

```

Allen Zweigen
Gemeinsames
herausziehen

- Wir ziehen die **continue**-Anweisungen aus den Zweigen hinter die Auswahl:

```

14      if (isdigit(copy.at(2))) {
16          copy = copy.substr(0, 2);
18      } else {
21          copy = copy.substr(0, 1);
23      }
        continue;
24      return true;

```

Nun ist deutlich zu erkennen, dass die auf **continue** folgende **return**-Anweisung in Zeile 24 unerreichbar ist. Allgemein gilt: Auf Sprunganweisungen statisch folgende Anweisungsfolgen werden nie ausgeführt, sie sind toter Code.

Tote Zweige entfernen

- Wir entfernen die unerreichbare **return**-Anweisung in Zeile 24.

Das Ergebnis dieser vier Änderungen an Programm 2 zeigt Programm 3.

Programm 3

C++: check vereinfacht

```

01  bool check (string way) {
02      while (true) {
03          if (way == "")
04              return true;
05          if (way.size() == 1)
06              return false;
07          if (way.at(0) == 'n' || way.at(0) == 'N' || way.at(0) == 's' || way.at(0) == 'S' ||
08              way.at(0) == 'e' || way.at(0) == 'E' || way.at(0) == 'w' || way.at(0) == 'W') {
09              if (isdigit(way.at(1))) {
10                  if (isdigit(way.at(2))) {
11                      way = way.substr(0, 2);
12                  } else {
13                      way = way.substr(0, 1);
14                  }
15                  continue;
16              } else
17                  return false;
18          } else
19              return false;
20      }
21  }

```

Als Nächstes stört die lange, aus acht Vergleichen disjunktiv kombinierte Bedingung

```

way.at(0) == 'n' || way.at(0) == 'N' || way.at(0) == 's' || way.at(0) == 'S' ||
way.at(0) == 'e' || way.at(0) == 'E' || way.at(0) == 'w' || way.at(0) == 'W'

```

in Zeile 10. Sie eignet sich zur funktionalen Zerlegung.

Zu große Teile
zerlegen

- Wir definieren für die Bedingung in Zeile 10 eine Funktion:

```

bool isGoodLetter (char c) {
    return
        c == 'n' || c == 'N' || c == 's' || c == 'S' || c == 'e' || c == 'E' || c == 'w' || c == 'W';
}

```

Da sie von vier Buchstaben den Klein- und den Großbuchstaben prüft, lässt sie sich mit der *tolower*-Funktion (oder *toupper*) der C-Standardbibliothek verkürzen:

```

bool isGoodLetter (char c) {
    c = tolower(c);
    return c == 'n' || c == 's' || c == 'e' || c == 'w';
}

```

Die Zerlegung führt von Programm 3 zu Programm 4.

Programm 4

C++: check zerlegt

```

    bool isGoodLetter (char c) {
        c = tolower(c);
        return c == 'n' || c == 's' || c == 'e' || c == 'w';
    }

01  bool check (string way) {
02      while (true) {
03          if (way == "")
04              return true;
05          if (way.size() == 1)
06              return false;
07          if (isGoodLetter(way.at(0))) {
08              if (isdigit(way.at(1))) {
09                  if (isdigit(way.at(2))) {
10                      way = way.substr(0, 2);
11              } else {
12                  way = way.substr(0, 1);
13              }
14              continue;
15          } else
16              return false;
17      } else
18          return false;
19  }
20  }

```

Ziel ist, die fünf Sprunganweisungen (vier **return**, ein **continue**) in Programm 4, die aus dem Schleifenrumpf hinausführen, zu eliminieren und so die Schleife von der unstrukturierten Form in eine strukturierte Form zu bringen. Dazu müssen die Zweige mit den **return**-Anweisungen nach oben, der Zweig mit der **continue**-Anweisung muss nach unten wandern.

return an den Anfang,
continue ans Ende der
Schleife bringen

- Wir negieren die Bedingungen der **if**-Anweisungen in den Zeilen 10 bis 30 und 12 bis 27 und vertauschen ihre **then**- und **else**-Zweige:

```

10      if (!isGoodLetter(way.at(0)))
30          return false;
28      else {
12          if (!isdigit(way.at(1)))
27              return false;
25          else {
14              if (isdigit(way.at(2))) {
16                  way = way.substr(0, 2);
18              } else {
21                  way = way.substr(0, 1);
23              }
24              continue;
26          }
27      }

```

Nun können wir das **else** in Zeile 28 und das **if** in Zeile 12 zusammenziehen und so ein Klammerpaar sparen und die Schachtelungstiefe um 1 mindern:

```

10    if (!isGoodLetter(way.at(0)))
30    return false;
28    else if (!isdigit(way.at(1)))
27    return false;
25    else {
14        if (isdigit(way.at(2))) {
16            way = way.substr(0, 2);
18        } else {
21            way = way.substr(0, 1);
23        }
24        continue;
    }

```

Dies in Programm 4 eingesetzt ergibt Programm 5.

Programm 5

C++: check mit
vertauschten Zweigen

```

    bool isGoodLetter (char c) {...}
01    bool check (string way) {
04        while (true) {
06            if (way == "")
07                return true;
08            if (way.size() == 1)
09                return false;
10            if (!isGoodLetter(way.at(0)))
30            return false;
28            else if (!isdigit(way.at(1)))
27            return false;
25            else {
14                if (isdigit(way.at(2))) {
16                    way = way.substr(0, 2);
18                } else {
21                    way = way.substr(0, 1);
23                }
24                continue;
            }
31        }
32    }

```

Die *continue*-Anweisung in Programm 5 ist deutlich überflüssig, weil der Ablauf von Zeile 24 ohnehin zu Zeile 04 führen würde. Sie ist auch in den Programmen 3 und 4 überflüssig, aber dort fällt das kaum auf. In Programm 2 sind die *continue*-Anweisungen nur wegen der unnötigen *return*-Anweisung in Zeile 24 erforderlich. Hätte lakers1230 auf dieses *return* verzichtet, so hätte er nicht um Hilfe rufen müssen.

continue eliminieren

- Weg mit dem unnützen *continue* in Zeile 24!

continue-Anweisungen lassen sich oft mit Ockhams Rasiermesser nach dieser Vorgehensweise entfernen. Da der *else*-Zweig von Zeile 25 nur noch eine Anweisung enthält, lässt sich das *else* mit dem *if* von Zeile 14 zusammenziehen, ein Klammerpaar und eine Schachtelung sparen. Dann sparen wir gleich noch die Klammern der Zweige in den Zeilen 16 und 21 und erhalten Programm 6, in dem die sechs Klammerpaare von Programm 1 auf zwei Paare geschrumpft sind.

Ein flüchtiger Blick auf Programm 5 oder Programm 6 erkennt eine Folge von drei *if*-Anweisungen, die ersten beiden einseitig, die dritte drei- bzw. vierseitig. Doch dieser Eindruck ist falsch – tatsächlich handelt es sich um eine fünf- bzw. sechseitige Auswahl, nur erschließt sich dies erst beim Lesen der ersten beiden *then*-Zweige und Erkennen der *return*-Anweisungen darin. Die Folge der *if-return*-Anweisungen verschleiert die Mehrfachauswahl und mindert so die Verständlichkeit. Also weg mit dem Schleier – die Mehrfachauswahl zeige sich unverhüllt!

Programm 6

C++: check ohne
continue

```

    bool isGoodLetter (char c) {...}
01  bool check (string way) {
04      while (true) {
06          if (way == "")
07              return true;
08          if (way.size() == 1)
09              return false;
10          if (!isGoodLetter(way.at(0)))
30              return false;
28          else if (!isdigit(way.at(1)))
27              return false;
25          else if (isdigit(way.at(2)))
16              way = way.substr(0, 2);
18          else
21              way = way.substr(0, 1);
31      }
32  }

```

Deutliche Mehrfachauswahl ist erstes Ziel; zweites Ziel ist, vermeidbare Unstetigkeiten zu eliminieren. Dazu suchen wir in Programm 6 Unstetigkeitsstellen, bei denen ein Semikolon zuviel die Semantik der Funktion stark verändert. In Zeile 04 würde



```
04  while (true); {
```

zu einer Endlosschleife führen, aber diese Unstetigkeitsstelle lässt sich nicht beseitigen. In Zeile 06 würde



```
06  if (way == "");
```

die Funktion stets **true** liefern lassen, in Zeile 08 würde



```
08  if (way.size() == 1);
```

die Funktion stets **false** liefern lassen. Diese Unstetigkeitsstellen sind vermeidbar: Sie verschwinden, wenn die einseitigen Auswahlanweisungen durch zweiseitige ersetzt werden. Dies ist möglich, weil am Ende ihrer Zweige Sprunganweisungen stehen.

if-return durch if-return-
else ersetzen

- Wir ersetzen die einseitigen **if**-Anweisungen in den Zeilen 06 und 08 durch zweiseitige **if-else**-Anweisungen.

Die zwei einseitigen Auswahlen gefolgt von der vierseitigen Auswahl in Programm 6 mutieren zu einer sechseitigen Auswahl in Programm 7.

Programm 7

C++: check mit
Sechsfachauswahl

```

    bool isGoodLetter (char c) {...}
01  bool check (string way) {
04      while (true) {
06          if (way == "")
07              return true;
08          else if (way.size() == 1)
09              return false;
10          else if (!isGoodLetter(way.at(0)))
30              return false;
28          else if (!isdigit(way.at(1)))
27              return false;
25          else if (isdigit(way.at(2)))
16              way = way.substr(0, 2);
18          else
21              way = way.substr(0, 1);
31      }
32  }

```

Von den sechs Zweigen in Programm 7 enthalten die ersten vier **return**-Anweisungen (Zeilen 07, 09, 30, 27), d.h. die ersten vier **if**-Bedingungen sind Abbruchbedingungen der **while**-Schleife (Zeilen 04 bis 31). Die drei gleichen Zweige mit

return false;

(Zeilen 08 bis 27) lassen sich zu *einem* Zweig zusammenfassen, indem man ihre **if**-Bedingungen disjunktiv mit dem kurz auswertenden Oder-Operator **||** verknüpft:

```
06    if (way == "")
07        return true;
08    else if (way.size() == 1 || !isGoodLetter(way.at(0)) || !isdigit(way.at(1)))
27        return false;
25    else if ...
```

Gleiche Zweige
zusammenfassen

- Wie konsolidieren gleiche Zweige durch „verodern“ ihrer Bedingungen.

Die Zweigkonsolidierung in Programm 7 liefert Programm 8.

Programm 8

C++: check mit
Vierfachauswahl

```
bool isGoodLetter (char c) {...}
01 bool check (string way) {
04     while (true) {
06         if (way == "")
07             return true;
08         else if (way.size() == 1 || !isGoodLetter(way.at(0)) || !isdigit(way.at(1)))
27             return false;
25         else if (isdigit(way.at(2)))
16             way = way.substr(0, 2);
18         else
21             way = way.substr(0, 1);
31     }
32 }
```

Mit den Abkürzungen

A $\equiv$ `way == ""`

B $\equiv$ `way.size() == 1 || !isGoodLetter(way.at(0)) || !isdigit(way.at(1))`

C $\equiv$ `if (isdigit(way.at(2))) way = way.substr(0, 2); else way = way.substr(0, 1);`

hat die Schleife in Programm 8 die unstrukturierte Form

Unstrukturiertes
Schleifenmuster

```
while (true) {
    if (A)
        return true;
    else if (B)
        return false;
    else
        C
}
```

Die Abbruchbedingungen *A* und *B* lassen sich disjunktiv zur „Gesamtabbruchbedingung“ $A \parallel B$ verknüpfen, die äquivalent zu $A \parallel (!A \ \&\& \ B)$ ist. Die kopfgesteuerte Bedingungsschleife in C++, die **while**-Schleife, verlangt aber die Fortsetzungsbedingung, also die negierte Abbruchbedingung $!A \ \&\& \ !B$. Nach der Schleife muss allerdings danach differenziert werden, ob sie wegen *A* oder *!A* (und deshalb *B*) beendet wurde.

Aufgabe 1

Schleifenumformung

Formen Sie das unstrukturierte Schleifenmuster in vier Schritten in ein strukturiertes Schleifenmuster um, wobei Sie die Korrektheit der Umformungsschritte mit Zusicherungen beweisen! $\square$

Anstelle der Umformung mit Zusicherungen kann eine Wahrheitstabelle helfen, die Situation zu verstehen.

Tabelle 1

Wahrheitstabelle

Abbruchbedingungen			Aktion
<i>A</i>	<i>B</i>	<i>!A && B</i>	
<i>false</i>	<i>false</i>	<i>false</i>	nicht abbrechen, weiter iterieren
<i>false</i>	<i>true</i>	<i>true</i>	<i>return false</i>
<i>true</i>	<i>false, true, undefiniert</i>	<i>false</i>	<i>return true</i>

Tabelle 1 zeigt: Wenn die Schleife terminiert, wird der Wert von *A* zurückgegeben. Also hat die strukturierte Bedingungsschleife die Form

Strukturiertes Schleifenmuster

```
while (!A && !B) {
    C
}
return A;
```

Bevor wir in diesem Muster *A*, *B*, *C* ersetzen, überlegen wir: Da stets *way.size() >= 0* gilt und wegen der Äquivalenz

A $\equiv$ *way* == "" $\equiv$ *way.size() == 0*

für die leere Zeichenkette gilt die Äquivalenzkette

```
way != "" && way.size() != 1                                      $\equiv$ 
way.size() > 0 && (way.size() == 0 || way.size() > 1)            $\equiv$ 
(way.size() > 0 && way.size() == 0) || (way.size() > 0 && way.size() > 1)  $\equiv$ 
false || way.size() > 1                                          $\equiv$ 
way.size() > 1.
```

Damit gilt

```
!A && !B                                                          $\equiv$ 
way != "" && way.size() != 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))  $\equiv$ 
way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1)).
```

Zudem ist in *C* die bedingte Anweisung durch einen bedingten Ausdruck ersetzbar:

```
C    $\equiv$    if (isdigit(way.at(2))) way = way.substr(0, 2); else way = way.substr(0, 1);
 $\equiv$    way = (isdigit(way.at(2)) ? way.substr(0, 2) : way.substr(0, 1));
 $\equiv$    way = way.substr(0, (isdigit(way.at(2)) ? 2 : 1));
```

Dies in das strukturierte Schleifenmuster eingesetzt ergibt

```
while (way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))) {
    way = way.substr(0, (isdigit(way.at(2)) ? 2 : 1));
}
return way.size() == 0;
```

Unstrukturierte Schleife strukturieren

- Wir ersetzen also die bedingten Sprünge aus dem Schleifenrumpf in den Zeilen 06 bis 27 durch eine Schleifenkopfbedingung.

Das Restrukturieren der Schleife in Programm 8 liefert Programm 9, in dem die sieben Sprunganweisungen im Schleifenrumpf von Programm 2 auf eine **return**-Anweisung am statischen Ende der Funktion *check* reduziert sind.

Programm 9

C++: check mit strukturierter Bedingungsschleife

```
bool isGoodLetter (char c) {...}
01  bool check (string way) {
04      while (way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))) {
25          way = way.substr(0, (isdigit(way.at(2)) ? 2 : 1));
31      }
      return way.size() == 0;
32  }
```

Nun wenden wir uns dem Ausdruck



`isdigit(way.at(2))`

in Zeile 25 zu. Die Fortsetzungsbedingung `way.size() > 1` in Zeile 04 garantiert, dass die Zeichenkette `way` wenigstens zwei Zeichen an den Positionen 0 und 1 enthält; sie garantiert aber nicht, dass ein drittes Zeichen an der Position 2 steht. Die Abfrage `at` der `string`-Klasse führt eine Indexprüfung durch; bei ungültigem Index wirft sie eine `out_of_range`-Ausnahme [C++10a]. Hat also `way` nur zwei Zeichen, so führt der Aufruf `check(way)` zu einer Ausnahme, die vom Aufrufer zu behandeln ist.

Wir wissen nicht, ob `lakers1230` dies gewollt hat, denn er hat keinen Hinweis darauf hinterlassen, was sein Programm 2 leisten soll: Er hat es nicht spezifiziert. Ohne Spezifikation können wir nicht über die Korrektheit einer Implementation reden.

3

Korrekturen

In Kapitel 2 haben wir Programm 2 schrittweise restrukturiert, also in funktional äquivalente Varianten umgeformt, ohne uns darum zu kümmern, welchem pragmatischen Zweck das Programm dienen soll. Nun stoßen wir beim Analysieren des Programmverhaltens auf Merkwürdigkeiten, die uns zum Nachdenken über die Pragmatik des Programms bringen. Wir können jedoch nur vermuten, was das Programm von `lakers1230` sinnvollerweise tun sollte, und die Vermutungen als Anforderungen formulieren.

Anforderung 1

Ausnahmen vermeiden

Die Funktion `check` löst keine `out_of_range`-Ausnahme aus.

Um Anforderung 1 zu erfüllen, ist vor dem Zugriff mit einem Index die Gültigkeit des Index zu prüfen, hier durch

`way.size() > 2 && isdigit(way.at(2)).`

Dies in Programm 9 eingesetzt liefert das funktional nicht äquivalente, aber pragmatisch sinnvollere Programm 10.

Programm 10

C++: check mit Indexprüfung

```
bool isGoodLetter (char c) {...}
01  bool check (string way) {
04      while (way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))) {
25          way = way.substr(0, (way.size() > 2 && isdigit(way.at(2)) ? 2 : 1));
31      }
      return way.size() == 0;
32  }
```

Um festzustellen, was Programm 10 leistet, verfolgen wir die Abläufe von Aufrufen `check(way)` mit verschiedenen aktuellen Parametern `way`, die alle Pfade abdecken.

Tabelle 2

Symbolische
Ablaufverfolgung zu
Programm 10

Aufruf <code>check(way)</code>		
Testfall	Eigenschaft von Parameter <code>way</code>	Ergebniswert
1	<code>size() == 0</code>	<i>true</i>
2	<code>size() == 1</code>	<i>false</i>
3	<code>size() > 1 && !isGoodLetter(way.at(0))</code>	<i>false</i>
4	<code>size() > 1 && isGoodLetter(way.at(0)) && !isdigit(way.at(1))</code>	<i>false</i>
5	<code>size() == 2 && isGoodLetter(way.at(0)) && isdigit(way.at(1))</code>	<i>false</i>
6	<code>size() > 2 && isGoodLetter(way.at(0)) && isdigit(way.at(1)) && !isdigit(way.at(2))</code>	<i>false</i>
7	<code>size() > 2 && isGoodLetter(way.at(0)) && isdigit(way.at(1)) && isdigit(way.at(2))</code>	<i>false</i>

Nach Tabelle 2 ist Programm 10 äquivalent zu Programm 11.

Programm 11
C++: check kurz

```
01  bool check (string w) {
      return w == "";
32  }
```

Entspricht Programm 11 den Intentionen von lakers1230? Hat er das 32 Zeilen lange Programm 1 geschrieben, um auszudrücken, was in eine Zeile passt? Vermutlich nicht. Aber was könnte lakers1230 beabsichtigt haben? Um Hypothesen über seine Absichten zu erhalten, untersuchen wir die Abläufe von Programm 10 in Tabelle 2 genauer.

Testfälle 1 bis 4

Die Testfälle 1 bis 4 sind leicht zu verfolgen, weil die Schleifenbedingung

```
way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))
```

beim ersten Mal **false** ist, der Schleifenrumpf also keinmal ausgeführt wird, die Schleife terminiert und

```
return way.size() == 0;
```

true oder **false** liefert.

Testfall 5

Bei Testfall 5 ist die Schleifenbedingung beim ersten Mal **true**, aber in Zeile 25 ist

```
way.size() > 2 && isdigit(way.at(2))  ==  false && undefiniert  ==  false,
```

sodass sich die Zuweisung

```
way = way.substr(0, 1)
```

ergibt. Was macht die Funktion *substr* der *string*-Klasse?

```
way.substr(position, length)
```

liefert die Teilzeichenkette von *way*, die beim Index *position* beginnt und maximal die Länge *length* hat, genauer diesen Ausschnitt [C++10s]:

```
way[position .. min(position + length - 1, way.size() - 1)]
```

Also ist *way.substr(0, 1)* die Zeichenkette, die nur aus dem Zeichen *way[0]* besteht. Die Zuweisung *way = way.substr(0, 1)* verkürzt *way* auf sein erstes Zeichen. Danach gilt *way.size() == 1*. Damit ist die Fortsetzungsbedingung in Zeile 04 wie bei Testfall 2 nicht mehr erfüllt.

Testfall 6

Testfall 6 verläuft ähnlich wie Testfall 5, nur gilt in Zeile 25

```
way.size() > 2 && isdigit(way.at(2))  ==  true && false  ==  false.
```

Testfall 7

Bei Testfall 7 gilt bei der ersten Iteration in Zeile 25

```
way.size() > 2 && isdigit(way.at(2))  ==  true && true  ==  true,
```

sodass sich die Zuweisung

```
way = way.substr(0, 2)
```

ergibt, die *way* auf seine ersten beiden Zeichen verkürzt. Damit gilt *way.size() == 2* und der Ablauf geht wie bei Testfall 5 weiter. Resümee der Analyse der Ablaufverfolgung von Programm 10:

Fazit

⊗ Die **while**-Schleife ist eine „Höchstens-zweimal-Schleife“.

⊗ *way* wird in der ersten Iteration auf ein oder zwei Anfangszeichen verkürzt, nochmal geprüft, und verworfen oder in der zweiten Iteration auf ein Zeichen verkürzt und dann verworfen.

Welchen Sinn ergibt das? Keinen. Vermutlich wollte lakers1230 *way* nicht auf seine Anfangszeichen verkürzen, sondern die geprüften Anfangszeichen von *way* abschneiden und mit dem Rest weiter iterieren. Diese Hypothese führt zu folgenden Anforderungen.

Anforderung 2

Schleife iterieren lassen

Die *while*-Schleife ist keine „Höchstens-zweimal-Pseudoschleife“, sondern eine echte Bedingungschleife, die einem Zweck dient.**Anforderung 3**

Parameter nicht auf geprüften Anfang kürzen

way wird nicht auf ein Anfangsstück verkürzt.**Anforderung 4**

Parameter auf nicht geprüftes Ende kürzen

In jeder Iteration werden die schon auf *isGoodLetter* und *isdigit* geprüften Anfangszeichen von *way* entfernt, sodass das ungeprüfte Endstück von *way* übrigbleibt.

Die Anforderungen 2 bis 4 werden erfüllt, wenn in Zeile 25 von Programm 10

```
way.substr(0, (way.size() > 2 && isdigit(way.at(2)) ? 2 : 1))
```

durch

```
way.substr(way.size() > 2 && isdigit(way.at(2)) ? 3 : 2)
```

ersetzt wird. Warum? Weil *substr* an Stelle des zweiten Parameters gemäß

```
way.substr(position)    ≡    way.substr(position, way.size())
```

den Standardwert einsetzt [C++10s]. Damit erhalten wir aus Programm 10 das funktional nicht äquivalente, aber pragmatisch sinnvollere Programm 12.

Programm 12

C++: check korrigiert

```

    bool isGoodLetter (char c) {...}
01  bool check (string way) {
04      while (way.size() > 1 && isGoodLetter(way.at(0)) && isdigit(way.at(1))) {
25          way = way.substr(way.size() > 2 && isdigit(way.at(2)) ? 3 : 2);
31      }
      return way.size() == 0;
32  }

```

Lässt sich Programm 12 spezifizieren? Ja, ganz leicht: *check* prüft, ob *way* den EBNF-Regeln

EBNF

```

Way      = { GoodLetter digit [ digit ] } .
GoodLetter = 'n' | 'N' | 's' | 'S' | 'e' | 'E' | 'w' | 'W' .

```

folgt. Es handelt sich um eine reguläre Sprache. Sie kann durch einen endlichen Automaten erkannt werden. Wie man einen endlichen Automaten konstruiert, der eine gegebene reguläre Sprache erkennt, ist Gegenstand des mki-B-Lehrmoduls *Theoretische Grundlagen 2*.**Aufgabe 2**

Lösung mit rekursivem Abstieg

Konstruieren Sie eine Lösung zu den EBNF-Regeln nach der Methode rekursiver Abstieg! Während die Algorithmen der Programme 1 bis 16 drei Zeichen vorausschauen, arbeitet der rekursive Abstieg mit einer Vorausschau (*lookahead*) von einem Zeichen. Was folgt daraus für die Algorithmen? □

Interpretation

Eine durch den Namen *way* nahe gelegte Interpretation der EBNF-Regeln ist: Die „guten Buchstaben“ stellen die vier Himmelsrichtungen north, south, east, west dar, die folgenden Ziffern eine Länge in der eingeschlagenen Richtung, *way* einen rechtwinkligen Polygonzug, den die Schildkröte der bekannten Turtle-Grafik abwandert.**4****Weitere Restrukturierungen**

Verständlichkeit

Nachdem das korrigierte Programm 12 etwas Sinnvolles leistet, bieten sich weitere Verbesserungen an. Zuerst verbessern wir die Verständlichkeit: *check* ist ein schlecht gewählter Name für eine boolesche Funktion, weil *check* ein Verb ist. Verben eignen sich als Namen für Prozeduren, die etwas tun, z.B. Zustände ändern. Dagegen model-

lieren boolesche Größen Eigenschaften, weshalb sie am besten mit Adjektiven, Partizipien oder Ähnlichem, auch Wörtern mit dem Präfix „ist“, benannt werden.

Klar benennen

- Wir benennen *check* um in *isWay*, *way* in *s*, und *isGoodLetter* in *isDirection*.

Dokumentieren

- Wir fügen die Spezifikation des Programms durch die EBNF-Regeln als Kommentar in den Quelltext ein.

Diese Schritte restrukturieren Programm 12 in Programm 13.

Programm 13

C++: *isWay* spezifiziert und verständlich

```
// Specification by EBNF-rules:
// Way      = { Direction digit [ digit ] } .
// Direction = 'n' | 'N' | 's' | 'S' | 'e' | 'E' | 'w' | 'W' .

bool isDirection(char c) {
    c = tolower(c);
    return c == 'n' || c == 's' || c == 'e' || c == 'w';
}

01 bool isWay(string s) {
04     while (s.size() > 1 && isDirection(s.at(0)) && isdigit(s.at(1))) {
25         s = s.substr(s.size() > 2 && isdigit(s.at(2)) ? 3 : 2);
31     }
    return s.size() == 0;
32 }
```

Effizienz

Wenn wir meinen, dass wir das Programm nicht mehr verständlicher, einfacher, besser strukturiert machen können, kümmern wir uns um seine Effizienz: Können wir den Programmablauf beschleunigen, ohne den Quelltext zu vernebeln?

Bei jeder Iteration wird beim Kürzen von *s* in Zeile 25 von Programm 13 eine Kopie des Endstücks der Zeichenkette angelegt, was laufzeitaufwändig ist. Das Kopieren der Zeichenkette lässt sich durch Verwenden einer Indexvariablen für die aktuelle Prüfposition vermeiden, wodurch auch das Kopieren des aktuellen Wertparameters überflüssig wird.

Kopieren sparen

- Wir ersetzen die Zuweisung an *s* in Zeile 25 durch das Erhöhen einer Indexvariablen *i*.

Referenz als
Parameter nutzen

- Da jetzt *s* in *isWay* unverändert bleibt, übernimmt es anstelle einer Kopie eines Objekts eine Referenz auf ein konstantes Objekt.

Damit wandelt sich Programm 13 zu Programm 14.

Programm 14

C++: *isWay* mit Index

```
// Specification by EBNF-rules: ...
bool isDirection(char c) { ... }

01 bool isWay(const string & s) {
    int i = 0;
04     while (i + 1 < s.size() && isDirection(s.at(i)) && isdigit(s.at(i + 1))) {
25         i += (i + 2 < s.size() && isdigit(s.at(i + 2)) ? 3 : 2);
31     }
    return i >= s.size();
32 }
```

5

Die Lösung

Zuletzt eine kosmetische Änderung: Der Zugriff auf die Zeichen von *s* ist nicht nur mit der Abfrage *s.at(position)* der *string*-Klasse möglich, sondern auch mit der von Reihungen gewohnten kürzeren Schreibweise *s[position]*, allerdings mit dem semantischen Unterschied, dass bei der Klammernotation der Index nicht auf Gültigkeit geprüft wird,

also Indexfehler zur Laufzeit unentdeckt bleiben [C++10b]. Da wir für Programm 14 nachweisen können, dass alle Indexwerte gültig sind, dürfen wir *at()* durch *[]* ersetzen.

Klammernotation bei
sicheren Zugriffen
verwenden

Programm 15

C++: *isWay* mit
Klammernotation

- Wir ersetzen in den Zeilen 04 und 25 *s.at(position)* durch *s[position]*.

Damit erhalten wir aus Programm 14 die letzte C++-Variante Programm 15.

```
// Specification by EBNF-rules:
// Way      = { Direction digit [ digit ] } .
// Direction = 'n' | 'N' | 's' | 'S' | 'e' | 'E' | 'w' | 'W' .

bool isDirection (char c) {
    c = tolower(c);
    return c == 'n' || c == 's' || c == 'e' || c == 'w';
}

01 bool isWay (const string & s) {
    int i = 0;
04     while (i + 1 < s.size() && isDirection(s[i]) && isdigit(s[i + 1])) {
25         i += (i + 2 < s.size() && isdigit(s[i + 2]) ? 3 : 2);
31     }
    return i >= s.size();
32 }
```

Nur die Zeilen 31 und 32 von Programm 1 blieben unberührt. Looping back to start of while loop? Is nich neetich, det macht die *while*-Schleife janz von alleine!

Eiffel

Hilferufe nach Sprungbefehlen wie *return*, *continue*, *break* lässt *Eiffel* unerhört verschallen, denn es kennt keine Sprungbefehle, erzwingt von vornherein das Denken in strukturierten Algorithmen, und verhindert so das Betreten der Fettnäpfchen-Programme 1 bis 8. Deshalb zum Schluss das zum C++-Programm 15 äquivalente *Eiffel*-Programm 16.

Programm 16

Eiffel: *is_way*

```
is_direction (c : CHARACTER) : BOOLEAN
-- EBNF-rule: Direction = 'n' | 'N' | 's' | 'S' | 'e' | 'E' | 'w' | 'W' .
do
    inspect c.as_lower
    when 'n', 's', 'e', 'w' then Result := True
    else
    end
end -- is_direction

01 is_way (s : STRING) : BOOLEAN
-- EBNF-rule: Way = { Direction digit [ digit ] } .
local
    i : INTEGER
do
    from
        i := 1
04    until
        i >= s.count or else not is_direction (s [i]) or else not s [i + 1].is_digit
    loop
        i := i + 2
14    if i <= s.count and then s [i].is_digit then
        i := i + 1
    end
31 end
    Result := i > s.count
32 end -- is_way
```

In Programm 16 ist *is_direction* mit der Mehrfachauswahlanweisung **inspect** implementiert, weil sie ohne lokale Variable auskommt, die sonst wegen der Schreibgeschütztheit des Parameters *c* benötigt würde. Der **else**-Zweig ist leer, weil *Result* mit *False* initialisiert ist.

Der Indexbereich von *Eiffel-STRING*-Objekten ist $1 \dots \text{count}$. Da *Eiffel* keine bedingten Ausdrücke kennt, steht im Schleifenrumpf von *is_way* eine bedingte Anweisung. Statt einer zweiseitigen Auswahl bietet sich an, die stets erforderliche Erhöhung des Index um 2 vor die Auswahl zu ziehen, sodass eine einseitige Auswahl übrig bleibt, die die Option für die zweite Ziffer deutlich ausdrückt.

A

Literatur

- [C++10a] The C++ Resources Network → C++ Reference → Strings library → string → at (2000–2010), <http://www.cplusplus.com/reference/string/string/at/> (Zugriff 2012-09-25)
- [C++10b] The C++ Resources Network → C++ Reference → Strings library → string → operator[] (2000–2010), [http://www.cplusplus.com/reference/string/string/operator\[\]/](http://www.cplusplus.com/reference/string/string/operator[]/) (Zugriff 2012-09-25)
- [C++10s] The C++ Resources Network → C++ Reference → Strings library → string → substr (2000–2010), <http://www.cplusplus.com/reference/string/string/substr/> (Zugriff 2012-09-25)
- [DIC10] Dream.In.Code: *Looping Back To Start Of While Loop*. Posted 19 October 2010 - 12:05 AM. MediaGroup1 LLC (2001–2010), Dream.In.Code → Programming Help → C and C++, <http://www.dreamincode.net/forums/topic/195689-looping-back-to-start-of-while-loop/> (Zugriff 2012-09-25)

B

Aufgabenlösungen

Lösung zu Aufgabe 1 Schleifenumformung

Das unstrukturierte Schleifenmuster von S. 9 ist mit Zusicherungen in ein strukturiertes Schleifenmuster umzuformen.

Unstrukturiertes
Schleifenmuster

```
while (true) {
    if (A)
        return true;
    else if (B)
        return false;
    else
        C
}
```

≡ // Zusicherungen einfügen

```
while (true) {
    if (A) {
        assert(A);
        assert(A == true);
        return true;
    } else if (B) {
        assert(!A && B);
        assert(A == false);
        return false;
    } else {
        assert(!A && !B);
        C
    }
}
```

≡ // **true** und **false** durch *A* ersetzen

```
while (true) {
  if (A) {
    assert(A);
    assert(A == true);
    return A;
  } else if (B) {
    assert(!A && B);
    assert(A == false);
    return A;
  } else {
    assert(!A && !B);
    C
  }
}
```

≡ // Gleiche Zweige zusammenfassen

```
while (true) {
  if (A || B) {
    assert(A || (!A && B));
    assert(A == true || A == false);
    return A;
  } else {
    assert(!A && !B);
    C
  }
}
```

≡ // Fortsetzungsbedingung zu **while**, **return** hinter die Schleife bringen

Strukturiertes
Schleifenmuster

```
while (!A && !B) {
  assert(!A && !B);
  C
}
assert(A || (!A && B));
return A;
```

Lösung zu Aufgabe 2
Lösung mit
rekursivem Abstieg

Der Lexer (Scanner) erhält *string s* als Eingabe, verbirgt es in seiner Implementation, und bietet dem Zerteiler (Parser) diese Schnittstelle:

void next () Aktion: Versucht, das nächste Zeichen von *s* zu lesen und weist es *symbol* zu oder setzt *endOfInput* auf **true**, wenn kein Zeichen mehr vorhanden ist.

char symbol Abfrage: zuletzt gelesenes Zeichen; Vorbedingung: *!endOfInput*.

bool endOfInput Abfrage: Ist die Eingabe *s* erschöpft?

Zur EBNF-Regel

// *Direction* = 'n' | 'N' | 's' | 'S' | 'e' | 'E' | 'w' | 'W'.

wird von Programm 15

```
bool isDirection (char c) {
  c = tolower(c);
  return c == 'n' || c == 's' || c == 'e' || c == 'w';
}
```

als nebeneffektfreie Prüffunktion für den Zerteiler übernommen. Zur EBNF-Regel

// *Way* = { *Direction digit* [*digit*] }.

ergibt sich die nebeneffektbehaftete Zerteilerfunktion:

Strukturierte Variante

```

bool isWay () {
    bool result = true;
    while (result && !endOfInput && isDirection(symbol)) {
        next();
        if (!endOfInput && isdigit(symbol)) {
            next();
            if (!endOfInput && isdigit(symbol)) {
                next();
            }
        } else {
            result = false;
        }
    }
    return result && endOfInput;
}

```

Die Vorausschau von nur einem Zeichen (*symbol*) führt verglichen mit Programm 15 zu einem tiefer geschachtelten Algorithmus (Schachtelungstiefe 4 statt 2), der zusätzlich zur *while*-Schleife eine zweiseitige und eine einseitige *if*-Anweisung benötigt. Die strukturierte Variante benötigt zudem die Steuervariable *result* zum Verlassen der Schleife. Dies ist ein Beispiel für einen strukturierten Algorithmus, der verglichen mit dem äquivalenten unstrukturierten Algorithmus

Unstrukturierte Variante

```

bool isWay () {
    while (!endOfInput && isDirection(symbol)) {
        next();
        if (!endOfInput && isdigit(symbol)) {
            next();
            if (!endOfInput && isdigit(symbol)) {
                next();
            }
        } else {
            return false;
        }
    }
    return endOfInput;
}

```

den Nachteil hat, eine zusätzliche Variable zu erfordern.