3 Steuerungsstrukturen

Perl lernen

ISBN 3-8273-1650-2

Was Sie in diesem Modul lernen

- Sie lernen die verschiedenen verfügbaren Bedingungsanweisungstypen kennen.
- Einige Steuerungsstrukturen, die innerhalb von Schleifen verwendet werden können, werden erklärt.
- Sie lernen geschachtelte Schleifen kennen.

Bisher haben Sie etwas über die verschiedenen Datentypen in Perl und über die Verwendung von Variablenzuweisungen und Operatoren gelernt. Jetzt lernen Sie etwas über Steuerungsstrukturen und ihre Verwendung im Programm. Diese Strukturen werden in den meisten Programmen eingesetzt und bieten Ihnen eine gewisse Kontrolle über die Programmausführung. In diesem Modul betrachten wir zuerst die elementarste Einheit einer Steuerungsstruktur, den Anweisungsblock. Dann beschäftigen wir uns mit den anderen Bedingungsanweisungen. Wenn Sie diesen Abschnitt beendet haben, zeigen wir Ihnen einige Steuerungsanweisungen, die Sie in einer Schleife verwenden können, und erläutern zum Abschluss geschachtelte Schleifen und deren Funktionsweise.

3.1 Bedingungsanweisungen

Bedingungsanweisungen werden in der Programmiersprache verwendet, wenn eine Entscheidung getroffen werden muss, bevor ein bestimmter Codeabschnitt ausgeführt wird. Vielleicht soll vorher sichergestellt werden, dass eine bestimmte Zahl erreicht wurde, oder eine Variable soll daraufhin überprüft werden, ob sie den richtigen Wert enthält. Bevor wir die verschiedenen Bedingungsanweisungen ausführlich behandeln, müssen wir ihren elementarsten Teil erläutern: den Anweisungsblock.

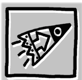
3.1.1 Anweisungsblöcke

Ein Anweisungsblock ist eine Folge von Anweisungen, die in geschweiften Klammern zusammengefasst werden. *Geschweifte Klammern* bestehen aus den beiden Zeichen { und }. Ein Anweisungsblock in einem Programm sieht in etwa so aus:

```
{
$a = 12;
$a += 5;
print "a equals $a\n";
}
```

Im folgenden Beispiel stehen die geschweiften Klammern an einer anderen Stelle:

```
{$a = 12;
$a += 5;
print "a equals $a\n";}
```



Obwohl die geschweiften Klammern an verschiedenen Stellen stehen, haben beide Anweisungsblöcke die gleiche Funktion. Sie können die geschweiften Klammern an jede beliebige Stelle setzen, solange die Codezeilen, die als Anweisungsblock zusammengefasst werden sollen, von ihnen eingeschlossen werden.

Die Codezeilen in einem Anweisungsblock werden immer ausgeführt, also könnte man sie auch *unbedingte Anweisung* nennen. Sie können zwar in Perl auch einen leeren Anweisungsblock verwenden, aber er ist ohne Nutzen.

Da Sie nun wissen, was Anweisungsblöcke sind und wie sie funktionieren, erläutern wir nun die verschiedenen Bedingungsanweisungstypen.

3.1.2 Die if-then-else-Bedingungsanweisung

Die if-then-else-Bedingungsanweisung prüft, ob ein Ausdruck true oder false ist. Ist das Ergebnis des Ausdrucks true, wird der then-Anweisungsblock ausgeführt. Ist das Ergebnis aber false, wird der else-Anweisungsblock ausgeführt. Im folgenden Beispiel sehen Sie die Syntax einer if-then-else-Bedingungsanweisung.

```
if (Ausdruck)
{
'then' Anweisungsblock
}
else
```

```
{
'else' Anweisungsblock
}
```

Wenn Sie den else-Abschnitt der Anweisung nicht benötigen, können Sie ihn auch weglassen. Hier sehen Sie eine if-then-Anweisung ohne else.

```
if (Ausdruck)
{
'then' Anweisungsblock
}
```

Zwischen einer if-then-else-Anweisung und einer if-then-Anweisung gibt es keinen syntaktischen Unterschied, außer dass der else-Abschnitt wegfällt.

Da Sie nun die Syntax einer if-then-else-Anweisung kennen, sehen wir uns ein kurzes Programmbeispiel dazu an.

```
#!/usr/bin/perl
$a = 20;
if ($a = 15){
print "a is equal to 15.\n";
}
else {
print "a is not equal to 15.\n";
}
```

Dieses Programm gibt „a is equal to 15“ aus, wenn der if-Ausdruck true ist, ansonsten lautet die Ausgabe „a is not equal to 15“.

Perl nimmt keinen else-Anweisungsblock ohne vorherige if-Anweisung an. Wenn nur die Fälle wichtig sind, in denen der Ausdruck false ist, können Sie die unless-Anweisung statt einer if-then-else-Anweisung verwenden. Die Syntax für eine unless-Bedingungsanweisung sieht folgendermaßen aus:

```
unless (Ausdruck)
{
'unless' Anweisungsblock
}
else
{
'else' Anweisungsblock
}
```

Die unless-Bedingungsanweisung berechnet für den Ausdruck einen true/false-Wert und führt den unless-Anweisungsblock aus, wenn der

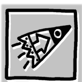
Ausdruck false ist. Ist der Ausdruck true, wird der else-Anweisungsblock ausgeführt. Der else-Teil dieser Anweisung kann wie bei der if-then-else-Anweisung weggelassen werden.

Die unless-Bedingungsanweisung ist besonders nützlich, wenn für Sie nur die Fälle interessant sind, in denen der Ausdruck false ist. Wenn Sie eine unless-Bedingungsanweisung verwenden müssen, müssen Sie möglicherweise keine else-Anweisung einfügen. Verwendet man else mit unless, handelt es sich im Grunde genommen um eine umgekehrte if-then-else-Anweisung, die Ihnen also keine Vorteile bringt.

Wenn Sie if-then-else-Bedingungsanweisungen verwenden, können Sie elsif-Anweisungen hinzufügen, wenn es nötig ist. Die Syntax einer if-then-else-Bedingungsanweisung mit einer elsif-Anweisung sieht so aus:

```
if (expression)
{
'then' statement block
}
elsif (expression)
{
'elsif' statement block
}
else
{
'else' statement block
}
```

elsif-Anweisungen werden verwendet, wenn mehr als ein Ausdruck ausgewertet werden muss, damit der richtige Anweisungsblock ausgeführt werden kann. Wie bei der if-then-else-Anweisung wird auch hier der Rest der Anweisung ignoriert, wenn ein beliebiger Abschnitt der Steuerungsstruktur ausgeführt wird. Sie können in einer if-then-else-Steuerungsstruktur so viele elsif-Anweisungen verwenden, wie Sie benötigen.



Beachten Sie, dass die elsif-Anweisung nicht „elseif“ geschrieben wird.

Da wir nun alle Möglichkeiten der if-then-else-Bedingungsanweisung behandelt haben, sehen wir uns einige Programmbeispiele zur Verwendung der Anweisungen an.

```
#Pfad zur Perl-Binärdistribution
#!/usr/bin/perl
$a = 15;        #Skalarvariablen-Zuweisung
```

```
#Der Ausdruck $a < 25 ist true.
if ($a < 25) {       #if-Bedingungsanweisung
#Dieser Anweisungsblock wird ausgeführt.
print "a is less than 25";
}
#Die else-Bedingungsanweisung und ihr Anweisungsblock werden
#übersprungen.
else  {          #else-Bedingungsanweisung
print "a is greater than or equal to 25";
}
```

Die folgende Abbildung zeigt die Bildschirmausgabe dieses Beispiels.

Abbildung 3.1

Dieses Beispiel sollte zeigen, wie eine if-then-else-Bedingungsanweisung funktioniert. Als Nächstes betrachten wir ein Beispiel für eine unless-Bedingungsanweisung.

```
#!/usr/bin/perl
$a = 15;     #Skalarvariablen-Zuweisung
#Dieser Ausdruck ist false. Daraufhin wird der unless-
#Anweisungsblock ausgeführt.
#Beachten Sie, dass sich der Ausdruck gegenüber dem
#vorherigen Beispiel verändert hat.
unless ($a > 25) {  #unless-Bedingungsanweisung
print "a is still less than 25";
}
```

Abbildung 3.2 zeigt die Bildschirmausgabe dieses Beispiels.

Abbildung 3.2

Nun haben Sie einen guten Überblick über die if-then-else-Bedingungsanweisungen und über die Optionen, die Sie dabei verwenden können. Als Nächstes behandeln wir eine andere häufig verwendete Bedingungsanweisung: die while-Anweisung.

Zwischentest

- Zählen Sie die möglichen Teile einer if-then-else-Bedingungsanweisung auf.[1]
- Welche Bedingungsanweisung sollten Sie verwenden, wenn für Sie nur der else-Fall einer if-then-else-Bedingungsanweisung von Bedeutung ist?[2]

3.1.3 Die while-Bedingungsanweisung

Die *while*-Bedingungsanweisung ist sehr nützlich, da sie es ermöglicht, einen Anweisungsblock so oft wie nötig auszuführen. Die while-Bedingungsanweisung funktioniert wie eine if-then-else-Bedingungsanweisung, in der ein Ausdruck mit true/false ausgewertet wird. Der einzige wichtige Unterschied ist, dass sie den Anweisungsblock ausführt, solange der Ausdruck true ist. Hier sehen Sie die Syntax für eine while-Bedingungsanweisung.

```
While (expression)
{
'while' statement block
}
```

1. If-Anweisung, elsif-Anweisung, Anweisungsblock und else-Anweisung.
2. Die unless-Bedingungsanweisung.

Wenn Sie diese Syntax sehen und über eine mögliche Verwendung nachdenken, fragen Sie sich vermutlich, wie und wann dieser Ausdruck jemals false sein kann. Hier sind die Autoinkrement- und Autodekrement-Operatoren aus dem letzten Kapitel sehr nützlich. Sie können diese Operatoren verwenden, um einen Zahlenwert bei jeder Ausführung des Anweisungsblocks für die while-Bedingungsanweisung zu erhöhen oder zu vermindern. Im folgenden Programmbeispiel wird das deutlicher.

```
#!/usr/bin/perl
$a = 15;            #Skalarvariablen-Zuweisung
#Der Anweisungsblock wird so lange ausgeführt, bis $a
#kleiner als 25 ist
while ($a < 25){        #while-Bedingungsanweisung
print "$a is still less than 25 \n";
#Jedes Mal, wenn der Anweisungsblock ausgeführt wird, erhöht
#der Autoinkrement-Operator den Wert von $a um eins
$a++;                   #Autoinkrement-Operator
}
```

In dieser Abbildung sehen Sie die Bildschirmausgabe dieses Beispiels.

Abbildung 3.3

Aus einer while-Anweisung kann man leicht eine Endlosschleife machen. Hätten wir z.B. im vorherigen Beispiel die Autoinkrement-Anweisung im Anweisungsblock weggelassen, dann hätten wir nun eine Endlosschleife. Der Wert von $a wäre immer kleiner als 25 gewesen. Es gibt bei der Programmierung viele Verwendungsmöglichkeiten für Endlosschleifen. Das beste Beispiel ist, wenn Sie ein Programm erstellen, das in einer while-Endlosschleife laufen soll und das System permanent auf Fehler überprüft. Da ein solcher Fehler die Schleife beenden würde, wüssten Sie sofort, dass ein Fehler aufgetreten ist.

Der Anweisungsblock einer while-Bedingungsanweisung kann unendlich oft ausgeführt werden, er kann aber auch niemals ausgeführt werden. Ist der Ausdruck bereits bei der ersten Wiederholung false, so wird die komplette Anweisung übersprungen.

Nach der while-Anweisung wollen wir uns mit einer ähnlichen Bedingungsanweisung beschäftigen.

Zwischentest

- Was ist der funktionelle Hauptunterschied zwischen einer if-then-else- und einer while-Bedingungsanweisung?[1]
- Welche Ihnen bekannten Operatoren werden meistens in einer while-Schleife verwendet?[2]

3.1.4 Die until-Bedingungsanweisung

Die *until*-Bedingungsanweisung ist das Gegenstück der while-Anweisung. Der until-Anweisungsblock wird solange ausgeführt, bis der Ausdruck false ist. Die Syntax für eine until-Bedingungsanweisung sieht so aus:

```
until (expression)
{
'until' statement block
}
```

Im folgendem Programmbeispiel können Sie sehen, wie die until-Bedingungsanweisung funktioniert.

```
#!/usr/bin/perl
$a = 15;     #Skalarvariablen-Zuweisung
#Der Anweisungsblock wird ausgeführt, bis $a größer als 25
#ist.
until ($a > 25){     #until-Bedingungsanweisung
print "The value of a is $a \n";
#Jedes Mal, wenn der Anweisungsblock ausgeführt wird, erhöht
#dieser Autoinkrement-Operator den Wert von $a um eins.
$a++;      #Autoinkrement-Operator
}
```

Die folgende Abbildung 3.4 zeigt die Bildschirmausgabe des Beispiels.

1. Die if-then-else-Anweisung kann keine Schleifen erzeugen, die while-Anweisungen sollen das.
2. Die Autoinkrement- und Autodekrement-Operatoren

```
Command Prompt
D:\>D:\Perl\book\figure3.3code.pl
15 is still less than 25
16 is still less than 25
17 is still less than 25
18 is still less than 25
19 is still less than 25
20 is still less than 25
21 is still less than 25
22 is still less than 25
23 is still less than 25
24 is still less than 25

D:\>
```

Abbildung 3.4

Im Beispiel für die while-Anweisung wurde der Anweisungsblock zehnmal ausgeführt. Mit der until-Anweisung und dem umgekehrten Ausdruck wird der Anweisungsblock elfmal ausgeführt. Denken Sie bitte an diesen Umstand, wenn Sie Code schreiben. Die beiden Beispiele führen nicht zum gleichen Ergebnis, auch wenn es so aussieht.

Auch das folgende Programmbeispiel verwendet eine Benutzereingabe mit der until-Bedingungsanweisung.

```
#!/usr/bin/perl
#Dem Benutzer wird eine Frage gestellt.
print "what is 2 + 2\n";  #Frage an den Benutzer
#Die Ausführung dieser Anweisung ist erst vollständig, wenn
#der Benutzer auf die Eingabetaste gedrückt hat.
#Diese Zeile liest die Eingabe des Benutzers ein und
#speichert sie in $a.
$a = <STDIN>;    #Speichere die Eingabe des Benutzers.
until ($a == 4){    #until-Bedingungsanweisung
#Dieser Anweisungsblock wird ausgeführt bis $a gleich 4 ist.
print "You know that is not right, try again \n";
$a = <STDIN>;    #erneute Eingabe des Benutzers
}
#Wenn der until-Ausdruck true ist und der Anweisungsblock
#übersprungen wird, wird diese Zeile ausgeführt.
print "yes, that should have been easier for you";
```

Hier sehen Sie die Bildschirmausgabe, die von diesem Programm erzeugt wird.

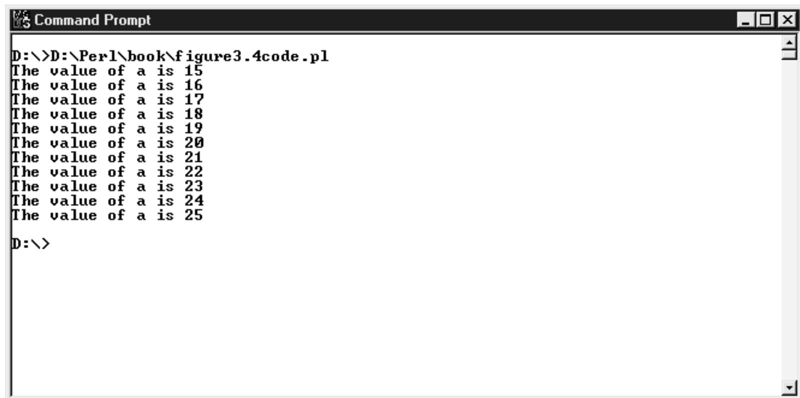


Abbildung 3.5

Dieses Beispiel zeigt, wie eine until-Anweisung verwendet werden kann, um die Ausführung eines Programms zu steuern. Das Programm fragt den Benutzer solange nach einer Antwort, bis er die richtige eingibt.

In der nächsten Bedingungsanweisung werden die while- und until-Anweisungen verwendet, allerdings in einer anderen Reihenfolge.

3.1.5 Die do while-until-Bedingungsanweisung

In den beiden vorhergehenden Abschnitten dieses Moduls ging es darum, dass bei einer while- oder until-Bedingungsanweisung die Anweisungsblöcke möglicherweise nie ausgeführt werden. Ist der Ausdruck false, wenn man true erwartet, wird die ganze Anweisung übergangen. In manchen Fällen soll der Anweisungsblock aber mindestens einmal ausgeführt werden. Mit der *do while-until*-Bedingungsanweisung können Sie das in Perl erreichen. Betrachten wir einmal die Syntax für eine solche Anweisung und bedenken Sie, dass die until-Anweisung die gleich Syntax hat, mit dem Unterschied, dass while durch until ersetzt wird.

```
do
{
'do' statement block
} while (expression)
```

Mit der do-while-until-Bedingungsanweisung kann der Anweisungsblock mindestens einmal ausgeführt werden, egal welches Ergebnis der Ausdruck hat. Das folgende Programmbeispiel verdeutlicht das.

```
#!/usr/bin/perl
$a = 12;    #Skalarvariablen-Zuweisung
#Die do while-Bedingungsanweisung
do  {       #do while-Bedingungsanweisung
#Der do while-Anweisungsblock
print "the statement block has been executed\n";
$a++;
}
#Dieser Ausdruck gibt an, wie oft der Anweisungsblock noch
#zusätzlich ausgeführt werden muss.
while ($a < 10)
```

In dieser Abbildung sehen Sie die Bildschirmausgabe nach der Ausführung dieses Beispiels.

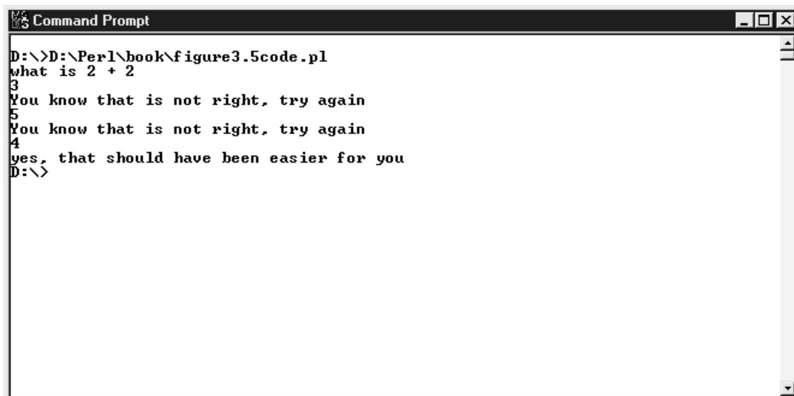


Abbildung 3.6

Wie Sie in der Abbildung sehen, wurde der Anweisungsblock einmal ausgeführt, obwohl der while-Ausdruck niemals true sein kann. Würden wir die while-Anweisung durch die until-Anweisung ersetzen, so erhielten wir eine Endlosschleife. Der Anweisungsblock würde solange ausgeführt, bis $a kleiner als zehn ist, was niemals der Fall sein kann.

Da Sie nun die do while-until-Steuerungsanweisung kennen, behandeln wir im Weiteren die for-Bedingungsanweisung.

3.1.6 Die for-Bedingungsanweisung

Die for-Anweisung wird von allen Programmierern verwendet, da mit der for-Anweisung kurz und knapp festgelegt werden kann, wie oft ein Anweisungsblock ausgeführt werden soll. Die Syntax für diese Bedingungsanweisung unterscheidet sich von den zuvor behandelten. Nach dem folgenden Beispiel werden wir die Einzelheiten erläutern.

```
for (statement; conditional expression; iterator statement)
{
'for' statement block
}
```

Der Syntax in der ersten Zeile entnehmen Sie, dass die Anweisung drei Teile hat. Die erste Zeile wird als „for Anweisung until Bedingungsanweisung; Iterator-Anweisung" gelesen. Meistens enthält der Ausdruck in der for-Anweisung einen Zahlenbereich, der angibt, wie oft der Anweisungsblock ausgeführt wird. Das folgende Beispiel verdeutlicht das.

```
#!/usr/bin/perl
#Hier erhalten Sie einen Wertebereich von eins bis neun.
for ($a=1; $a < 10; $a++){ #for-Bedingungsanweisung
print "the statement block has been executed $a times \n";
}
```

Diese Abbildung zeigt die Bildschirmausgabe, die von diesem Programmbeispiel erzeugt wird.

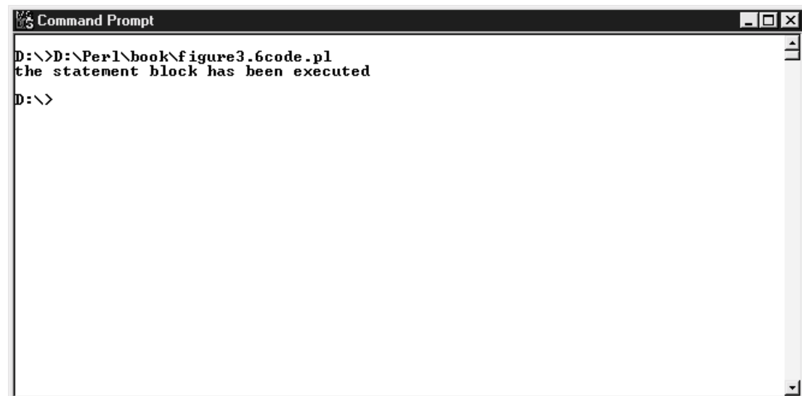


Abbildung 3.7

Die for-Anweisung aus dem vorherigen Beispiel wird als „solange $a gleich 1 bis $a kleiner 10, inkrementiere $a" gelesen. Der Wert von $a ist eins, wenn die for-Anweisung das erste Mal ausgeführt wird. Bevor der Anweisungsblock ausgeführt wird, wird der Wert von $a um eins

erhöht, sodass der neue Wert von $a nach der Blockausführung überprüft werden kann, um sicherzustellen, dass er kleiner als zehn ist. Ergibt die Überprüfung true, so wird $a wiederum um eins erhöht und der Anweisungsblock ausgeführt. In diesem Beispiel wurde der Anweisungsblock zehnmal ausgeführt.

- Was war der endgültige Wert von $a im vorherigen Programmbeispiel?[1]
- Wie oft wäre der Anweisungsblock ausgeführt worden, wenn der zweite Ausdruck $a <= 10 gelautet hätte, und was wäre der endgültige Wert von $a gewesen?[2]

Aufgabe 3.1 Eine Zählschleife erzeugen

Schreiben Sie ein Programm, das eine Zählschleife wie in dem vorherigen Programmbeispiel erzeugt. Sie können mit Ausnahme der for-Anweisung alle Bedingungsanweisungen verwenden, die Sie in diesem Modul bereits gelernt haben. Das Programm sollte anzeigen, wie oft der Anweisungsblock ausgeführt wurde und was der endgültige Wert der verwendeten Zählvariable ist.

Wenn Sie die for-Bedingungsanweisung verwenden und in der Anweisung, im Bedingungsausdruck oder in der Iterator-Anweisung mehr als eine Anweisung enthalten sein soll, können Sie ein Komma „,“ verwenden, um diese zu trennen. Dies ist auch bei der Verwendung mehrerer Komponenten möglich. Das folgende Programmbeispiel verdeutlicht dies.

```
#!/usr/bin/perl
#Beachten Sie, dass der Iterationsbereich der for-
#Bedingungsanweisung zwei durch ein Komma getrennte
#Anweisungen enthält.
#Die Ausgabeanweisung wird zusammen mit dem Autoinkrement-
#Operator ausgeführt.
for ($a = 1; $a < 10; $a++, print "the value of a is now $a \n"){
print "the statement block has been executed $a times \n";
}
```

Die folgende Abbildung 3.8 zeigt die Bildschirmausgabe, die von diesem Programm erzeugt worden ist.

---

1. – 10
2. – 10, 11

Abbildung 3.8

Dieses Programmbeispiel verfolgt, wie oft der Anweisungsblock ausgeführt wurde und welchen Wert $a jeweils hat. Jetzt haben wir die for-Bedingungsanweisung ausführlich behandelt und erläutern nun die letzten beiden Bedingungsanweisungen.

3.1.7 Die foreach-Bedingungsanweisung

Die *foreach*-Bedingungsanweisung kann durch den Inhalt eines Variablen-Arrays iterieren. Sie können eine Schleife auf diese Weise verschiedene Wertebereiche durchlaufen lassen, anstatt nur den Start- und den Endpunkt festzulegen.

Die Array-Variable haben wir bisher nicht besprochen. Stellen Sie sich vorläufig vor, dass die Array-Variable Zahlen und Zeichenketten enthält, die miteinander in Verbindung stehen. Diese Skalardaten nennt man *Elemente* des Arrays. Dieses Thema wird ausführlich im nächsten Modul behandelt.

Sehen wir uns zuerst die Syntax einer foreach-Bedingungsanweisung an, bevor wir uns ausführlicher damit beschäftigen.

```
foreach scalar variable (array variable)
{
'foreach' statement block
}
```

Die Skalarvariable in der foreach-Anweisung wird mit dem aktuellen Element des Arrays gleichgesetzt. Beachten Sie, dass nach der Ausführung der foreach-Anweisung die Skalarvariable wieder auf den ursprünglichen Wert zurückgesetzt wird. Eine Variable, die nur in bestimmten Programmabschnitten gültig ist, hat in diesen Abschnit-

ten ihren *Gültigkeitsbereich*. Da die Skalarvariable nur innerhalb der foreach-Schleife Gültigkeit hat, gilt der innerhalb der Schleife zugewiesene Wert nicht außerhalb der Schleife. Da dies ein wenig verwirrend klingt, geben wir das folgende Programmbeispiel zum besseren Verständnis an.

```
#!/usr/bin/perl
#Zuweisung der Skalarvariablen bevor in die foreach-Schleife
#eingetreten wird.
$b = 10;
#Gib den Wert von $b vor der Schleife auf dem Bildschirm
#aus.
print "the value of b is $b \n";
#Dies ist eine Array-Zuweisung. Das Array entspricht der
#Liste von Skalarzahlen auf der rechten Seite.
@a = (1,3,4,7,9,12);         #Array-Zuweisung
#Diese foreach-Anweisung verwendet die Array-Variable @a und
#die Skalarvariable $b.
foreach $b (@a){             #foreach-Bedingungsanweisung
#Dies ist der foreach-Anweisungsblock.
print "the value of b inside the loop is $b \n";
}
#Hier wird der Wert von $b nach der Schleife ausgegeben.
print "the value of b after the loop has executed is $b \n";
```

Diese Abbildung zeigt die Bildschirmausgabe des Programms.

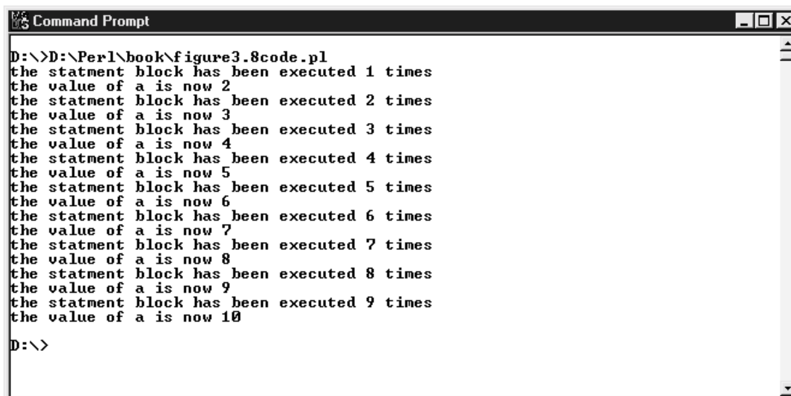


Abbildung 3.9

Ihnen sollte auffallen, dass der Wert von $b vor und nach der Schleife identisch ist, obwohl der Variablen innerhalb der foreach-Schleife andere Werte zugewiesen wurden. Dies liegt wiederum daran, dass die innerhalb der Schleife zugewiesenen Werte von $b außerhalb der Schleife keine Gültigkeit haben.

Um die foreach-Anweisung etwas einfacher zu gestalten, können Sie die implizierte Skalarvariable $_ verwenden. Die Syntax der foreach-Anweisung lautet dann:

```
foreach (array variable)
{
'foreach' statement block
}
```

Innerhalb des Anweisungsblocks können Sie die print-Anweisung verwenden, um den Wert des Array-Elements zu erhalten. In diesem Programmbeispiel verwenden wir die implizierte Skalarvariable $_.

```
#!/usr/bin/perl
#Array-Zuweisung
@a = (1,3,4,7,9,12);
#Die foreach-Bedingungsanweisung, die die implizierte
#Skalarvariable $_ verwendet
foreach (@a){
#Hier wird der Inhalt der Variablen $_ ausgegeben.
print;
print "\n";
}
```

Diese Abbildung zeigt die Bildschirmausgabe des Programms.

Abbildung 3.10

Da wir die implizierte Skalarvariable $_ verwenden, weiß Perl automatisch, dass Sie sich auf diese Variable beziehen, wenn Sie in der print-Anweisung keine Parameter angeben. Hierbei handelt es sich um die Kurzversion einer foreach-Anweisung, die beim Programmieren Zeit sparen soll.

Die letzte Bedingungsanweisung, die wir in diesem Modul behandeln, verwendet die meisten der Ihnen bereits bekannten Anweisungen und wiederholt sie in einer einfacheren Form.

3.1.8 Die einzeilige Bedingungsanweisung

Die *einzeilige* Bedingungsanweisung kann verwendet werden, wenn Sie innerhalb eines Anweisungsblocks nicht mehrere Anweisungen verwenden möchten. Wenn Sie nur eine Anweisung verwenden, spart Ihnen dies bei der Programmierung Zeit. Hier sehen Sie die Syntax für diese Anweisungen:

```
any Perl statement (space) if, while, until, or unless (space) expression
```

Und hier sehen Sie die Verwendung von verschiedenen einzeiligen Bedingungsanweisungen in einem Programm:

```
#!/usr/bin/perl
$a = 15;
$b = 10;
#Dies ist eine einzeilige if-Bedingungsanweisung.
$a++ if ($b < 20);
print "a equals $a \n";
print "a was incremented by one \n" if ($a == 16);
#$b wird so lange erhöht, wie es kleiner als 20 ist.
$b++ while ($b < 20);
print "b was incremented until it reached 20 \n" if ($b == 20);
#$a wird verringert, bis es gleich 2 ist.
$a  until ($a == 2);
print $a;
```

Die folgende Abbildung zeigt die Bildschirmausgabe, die von diesem Programm erzeugt wird.

Abbildung 3.11

Wie Sie an diesem Beispiel sehen, können Sie mit den einzeiligen Bedingungsanweisungen gegenüber der Implementierung von vollständigen Bedingungsstrukturen Zeit sparen.

Da Sie nun alle Bedingungsanweisungen in Perl kennen, behandeln wir im nächsten Abschnitt dieses Moduls Steuerungsanweisungen, die innerhalb von Programmschleifen verwendet werden können.

3.2 Steuerungsanweisungen

Perl enthält drei verschiedene Steuerungsanweisungen, die das normale Verhalten innerhalb einer Schleife außer Kraft setzen. Diese Anweisungen sind recht unkompliziert und wir können ohne Einleitung beginnen.

3.2.1 Die Steuerungsanweisung next

Die Steuerungsanweisung *next* kann in einer Schleife verwendet werden, um den Rest des Blocks während einer Iteration zu überspringen und bis zum normalen Schleifenende fortzufahren. Die Syntax für diese Anweisung sieht so aus:

```
next;
```

Im folgenden Programmbeispiel werden die while-Schleife und die Steuerungsanweisung next verwendet.

```
#!/usr/bin/perl
$a = 15;
#Dies ist eine elementare while-Schleife, wie wir sie im
#vorherigen Abschnitt behandelt haben.
while ($a < 25) {
$a++;
#Wenn $a gleich 20 ist, wird die Ausgabeanweisung
#übersprungen und die Schleife springt zu ihrer nächsten
#Iteration.
next if ($a == 20);
print "$a is still less than 25 \n";
}
print "exiting";
```

Abbildung 3.12 zeigt die Bildschirmausgabe nach der Ausführung dieses Programms.

```
D:\>D:\Perl\book\figure3.11code.pl
a equals 16
a was incremented by one
b was incremented until it reached 20
2
D:\>
```

Abbildung 3.12

Sie können in dieser Abbildung sehen, dass die print-Anweisung in der Schleife für $a = 20 übersprungen wurde und die Schleife bis zum normalen Abschluss ausgeführt wurde.

Im vorherigen Programmbeispiel haben wir eine einzeilige if-Bedingungsanweisung mit der Steuerungsanweisung next verwendet, um festzulegen, wann sie ausgeführt werden soll. Hätten wir nur die Steuerungsanweisung next verwendet, so wäre die print-Anweisung im Anweisungsblock bei jeder Iteration der Schleife übersprungen worden.

Dieses Beispiel sollte Ihnen ein Verständnis für die Steuerungsanweisungsfunktion next vermitteln. Deshalb möchten wir nun eine andere Steuerungsanweisung erläutern, die in einer Programmschleife verwendet werden kann.

3.2.2 Die Steuerungsanweisung last

Die Steuerungsanweisung *last* ähnelt der Steuerungsanweisung next mit dem Unterschied, dass sie die Schleife nach diesem Durchlauf beendet. Die Syntax für diese Anweisung ist einfach:

```
last;
```

Im nächsten Programmbeispiel sehen Sie, wie die Steuerungsanweisung last arbeitet.

```
#!/usr/bin/perl
$a = 15;
#Wieder verwenden wir eine elementare while-Schleife.
while ($a < 25) {
$a++;
#Wenn $a gleich 20 ist, wird die letzte Anweisung
```

```
#ausgeführt, die den Rest der Schleife beendet.
last if ($a == 20);
print "$a is still less than 25 \n";
}
print "exiting";
```

Diese Abbildung zeigt die Bildschirmausgabe nach der Ausführung des Programms.

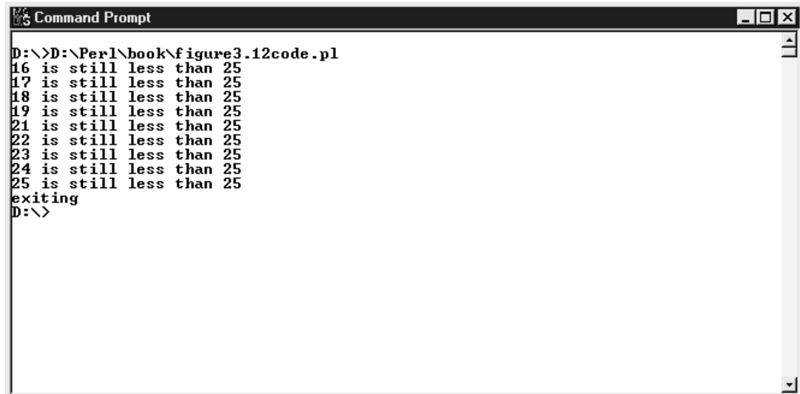


Abbildung 3.13

In dieser Abbildung sehen Sie, dass die while-Schleife beendet wurde, als $a = 20 war. Die letzte Anweisung setzt den Rest der Schleife außer Kraft. Nun müssen wir uns noch mit einer Steuerungsanweisung beschäftigen.

Wie im Programmbeispiel zu *next* haben wir auch hier eine einzeilige if-Bedingungsanweisung verwendet, um festzulegen, wann die Steuerungsanweisung last ausgeführt werden soll. Hätten wir die Steuerungsanweisung last allein verwendet, wäre $a einmal erhöht und das Programm dann beendet worden.

3.2.3 Die Steuerungsanweisung redo

Die Steuerungsanweisung *redo* wiederholt eine Iteration der Schleife. Die Syntax dieser Steuerungsanweisung sieht wie folgt aus:

```
redo;
```

Im folgenden Programmbeispiel sehen Sie die Verwendung dieser Steuerungsanweisung.

```
#!/usr/bin/perl
$a = 15;
```