PEARSON Education

ADDISON-WESLEY

Frank Eller

workshop delphi 6

ADDISON-WESLEY

An imprint of Pearson Education

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

3 Kontrollstrukturen

Alleine mit Daten und Anweisungen kann kein Programm funktionieren. Der Programmierer muss auch die Möglichkeit haben, Reaktionen abhängig von einer Bedingung durchzuführen, z.B. entsprechend des Werts einer Variablen eine bestimmte Aktion durchzuführen. Ein Beispiel hierfür ist die Division durch Null, die ja mathematisch gesehen nicht erlaubt ist. Aus diesem Grunde muss eine Kontrollmöglichkeit gegeben sein, um die Division zu verhindern, wenn der Divisor den Wert 0 hat.

Eine weitere Möglichkeit, die dem Programmierer offenstehen muss, ist die Möglichkeit, einen Programmteil ggf. wiederholen zu können. Es wäre sehr ungünstig, einen Programmteil, der sich mehrfach wiederholt, immer wieder programmieren zu müssen.

Wie jede andere Programmiersprache bietet auch Delphi hierfür einige Kontrollstrukturen, mit denen dem Programmierer alle Möglichkeiten, die irgendetwas mit Kontrolle zu tun haben, offenstehen. Delphi bietet verschiedene Arten von Schleifen für sich wiederholende Programmkonstruktionen und Verzweigungsmöglichkeiten für Entscheidungsfälle an.

3.1 Schleifen

Wie angesprochen dienen Schleifen der Wiederholung eines Programmteils oder auch nur einer Anweisung. Die Anzahl der Wiederholungen kann dabei variiert werden, es ist sowohl möglich, eine feste Anzahl von Wiederholungen zu programmieren, als auch die Wiederholung von einer Bedingung abhängig zu machen.

Die for-Schleife

Die `for`-Schleife ermöglicht es, eine bestimmte Anzahl von Durchläufen zu programmieren. Dabei wird eine so genannte Laufvariable zu Hilfe genommen, deren Datentyp ein ordinaler Typ sein muss – am häufigsten wird hier der Datentyp `integer` verwendet – und die bei jedem Schleifendurchlauf um eins hoch- oder heruntergezählt wird. Ist der angegebene Endwert erreicht, wird die Schleife beendet und mit der ersten nachfolgenden Anweisung weiter gemacht.

Die zu wiederholenden Anweisungen werden durch die reservierten Wörter `begin` und `end` eingekapselt. Sollte es sich nur um eine Anwei-

sung handeln, können `begin` und `end` auch entfallen. Die Syntax der `for`-Schleife sieht folgendermaßen aus:

Syntax

```
for <Laufvariable>:= <StartWert> to <Endwert> do
begin;
   //Liste von Anweisungen, die ausgeführt
   //werden sollen
end;
```

An Stelle des reservierten Worts `to` kann auch das reservierte Wort `downto` stehen. In diesem Fall zählt die Schleife rückwärts. Die Syntax bleibt die gleiche:

```
for <Laufvariable>:= <StartWert> downto <Endwert> do
begin;
   //Liste von Anweisungen, die ausgeführt
   //werden sollen
end;
```

Abbildung 3.1 zeigt das Ablaufschema einer `for`-Schleife.

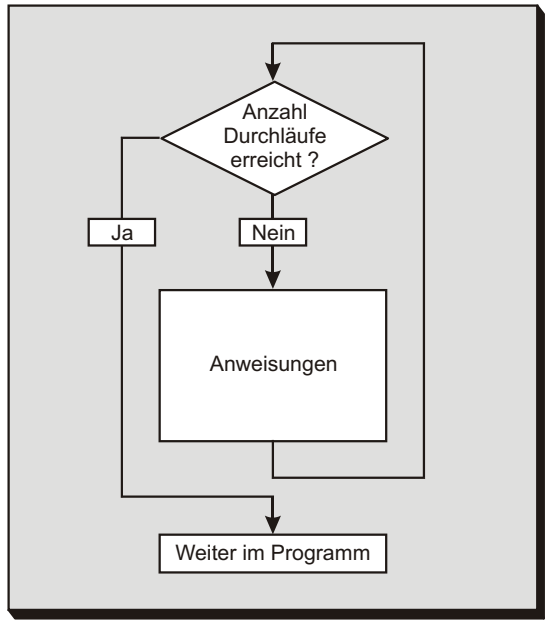


Abbildung 3.1: Das Ablaufschema einer for-Schleife

Mithilfe der `for`-Schleife lassen sich Wiederholungen also sehr einfach programmieren. Start- und Endwert können frei gewählt werden, auch negative Werte sind erlaubt, wenn die Laufvariable einen entsprechenden Datentyp besitzt. Ein Beispiel für eine `for`-Schleife wäre das Zusam-

menzählen aller Zahlen von 1 bis 100, eine Aufgabe, die Friedrich Gauss seinerzeit als Grundschüler in ca. zwei Minuten gelöst hat. Für die meisten von uns ist es einfacher, ein kleines Programm zu schreiben, das die Berechnungen durchführt.

```
function CalcGauss(nrFrom, nrTo: integer): integer;
var
  i: integer; //Die Laufvariable
begin;
  Result := 0;
  for i:=nrFrom to nrTo do
    inc(Result,i);
end;
```

In dieser Schleife passiert eigentlich nichts Weltbewegendes. Die Variable `Result` wird auf 0 gesetzt, also initialisiert, und dann jeweils um den Wert erhöht, der aktuell in `i` steht. Das ist exakt das, was wir zur Lösung unseres Problems benötigen. Die Prozedur `inc` erhöht die im ersten Parameter angegebene Variable um den Wert, den der zweite Parameter hat. Eine sehr häufig verwendete Prozedur.

Wenn Sie einen Debug-Lauf in Ihrem Programmtext durchführen, werden Sie unter Umständen feststellen, dass die `for`-Schleife immer rückwärts zählt, auch wenn Sie für das Vorwärtszählen programmiert wurde. Das ist keineswegs ein Bug, sondern ein Feature: es geht einfach schneller. Verantwortlich für dieses Verhalten ist die Compiler-Optimierung, die Sie zwar in den Programmoptionen ausschalten können, aber nicht sollten.

Die while-Schleife

Kommen wir jetzt zu einer Schleifenform, die keine Laufvariable benötigt, dafür aber eine Bedingung, deren Status bestimmt, ob die Schleife weiterhin durchlaufen wird oder nicht. Die `while`-Schleife wird so lange durchlaufen, wie die Bedingung, die im Schleifenkopf kontrolliert wird, wahr ist.

Auch hier können mehrere Anweisungen verwendet werden, die wieder zwischen `begin` und `end` gekapselt werden. Und ebenso wie bei der `for`-Schleife ist es auch hier wieder möglich, bei nur einer einzigen Anweisung `begin` und `end` wegzulassen.

Eine solche Schleifenform nennt man übrigens auch *abweisende Schleife*. Der Grund hierfür ist, dass für den Fall, dass die Bedingung bereits beim Beginn der Schleife falsch ist, kein Durchlauf stattfindet,

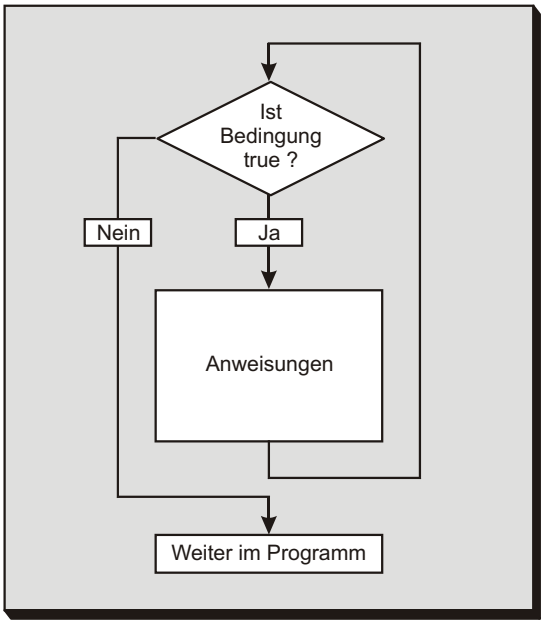
sondern das Programm direkt zur ersten Anweisung nach der Schleife springt, die Schleife also »abweist«. Die Syntax der `while`-Schleife sieht wie folgt aus:

Syntax

```
while <Bedingung> do
begin;
  //Anweisungen werden ausgeführt,
  //solange die Bedingung wahr ist
end;
```

Abbildung 3.2 zeigt das Ablaufschema einer `while`-Schleife.

Abbildung 3.2: Das Ablaufschema der while-Schleife



Bei der Bedingung kann es sich um eine boolesche Variable oder um einen Ausdruck handeln, der einen booleschen Wert ergibt, also einen Vergleichsausdruck. Das folgende Beispiel zeigt die Verwendung der `while`-Schleife.

```
function GGT(value1, value2: integer): integer;
var
  ggtReached: boolean;
  helpValue : integer;
begin;
  helpValue := value1;
  while (helpvalue>1) do
```

```
  begin;
    if ((value1 mod helpval)=0) and
       ((value2 mod helpval)=0) then
      break;
    dec(helpVal);
  end;
  Result := helpval;
end;
```

Die Funktion berechnet, wie aus dem Namen bereits ersichtlich, den größten gemeinsamen Teiler (ggT) zweier Zahlen. Die Abbruchbedingung der Schleife ist logisch zwingend der Wert der Variablen `helpValue`. Ist dieser gleich 1, wird die Schleife beendet (in diesem Fall gibt es keinen ggT). Andernfalls, wenn wir einen ggT finden, wird die Schleife beendet. Zu der dafür zuständigen Anweisung `break` und zu der benutzten Verzweigung mittels der `if`-Anweisung kommen wir gleich.

In diesem Beispiel wurde als Abbruchbedingung eine Vergleichsanweisung benutzt. Die Klammern stehen hier übrigens nur zur besseren Lesbarkeit und stellen eine Angewohnheit meinerseits dar – sie haben keinerlei Einfluss auf die Ausführungsgeschwindigkeit. Mögliche Vergleichsoperatoren finden Sie in Tabelle 3.1.

Operator	Bedeutung
>	Vergleich, ob ein Wert größer ist als der andere
<	Vergleich, ob ein Wert kleiner ist als der andere
>=	Vergleich, ob ein Wert größer oder gleich einem anderen ist
<=	Vergleich, ob ein Wert kleiner oder gleich einem anderen ist
=	Vergleich, ob zwei Werte identisch sind
<>	Vergleich auf Ungleichheit zweier Werte
in	Der `in`-Operator vergleicht, ob ein Wert in einer Menge enthalten ist.
is	Der `is`-Operator vergleicht, ob eine Variable von einem bestimmten Datentyp ist.

Tabelle 3.1: Die Vergleichsoperatoren von Delphi

Auf die Operatoren `is` und `in` kommen wir später noch zu sprechen.

Es muss sich bei der Bedingung nicht um einen einzelnen Vergleich handeln, es können auch mehrere verknüpft werden. Dann müssen sie allerdings in Klammern gesetzt werden, um den Zusammenhang sichtbar zu machen. Zum Verknüpfen von Bedingungen bzw. Ausdrücken können Sie die Operatoren verwenden, die in Tabelle 3.2 aufgelistet werden.

Tabelle 3.2: Die möglichen Verknüpfungen in Delphi

Verknüpfung	Bedeutung
and	Die gesamte Bedingung ist nur dann wahr, wenn alle Einzelbedingungen wahr sind.
or	Die gesamte Bedingung ist dann wahr, wenn eine der Einzelbedingungen wahr ist.
not	Der `not`-Operator negiert die Bedingung, aus wahr wird falsch und aus falsch wird wahr.

Wenn Sie die angesprochenen Klammern nicht verwenden, liefert Delphi einen Fehler und verweigert die Kompilierung. Der Grund hierfür ist, dass die Operatoren in Tabelle 3.2 nicht nur zum logischen Verknüpfen verwendet werden, sondern auch eine binäre Verknüpfung durchführen.

Ein recht leicht verständliches Beispiel für eine solche logische Operation, z.B. die `not`-Operation, wäre das Umschalten eines booleschen Werts bei jedem Klick auf ein Steuerelement. Auf diese Art und Weise könnten Sie z.B. eine Symbolleiste ein- oder ausblenden:

```
procedure MenuItem1Click(Sender: TObject);
begin;
  ToolBar1.visible := not ToolBar1.visible;
end;
```

Durch diese eine Zeile wird der Wert der Eigenschaft `visible` ständig negiert, aus `false` wird `true` und umgekehrt. Und da `visible` die Sichtbarkeit der Toolbar (oder einer beliebigen anderen Komponente) festlegt, wird diese mit jedem Klick auf den entsprechenden Menüpunkt ein- oder ausgeblendet.

Die repeat-Schleife

Die `repeat`-Schleife wird ebenfalls abhängig von einer Bedingung durchlaufen. Der Unterschied zur `while`-Schleife besteht darin, dass die Bedingung erst am Ende der Schleife kontrolliert wird. Damit ergibt sich, dass die Anweisungen innerhalb einer `repeat`-Schleife mindestens einmal durchlaufen werden. Eine solche Schleife nennt man auch *nicht-abweisende Schleife*, denn sie wird in jedem Fall einmal durchlaufen. Das ist auch der Grund, warum es zwei derartige Schleifen in Delphi gibt. Je nach Situation kann entweder die `while`-Schleife oder die `repeat`-Schleife das richtige Verhalten an den Tag legen.

Bei der Syntax fällt auf, dass die `repeat`-Schleife keinen Gebrauch von `begin` und `end` macht. Alle Anweisungen, die zwischen `repeat` und

`until` stehen, sind Bestandteil der Schleife. Die Syntax der `repeat`-Schleife sieht folgendermaßen aus:

Syntax

```
repeat
   //Anweisungsliste für die repeat-Schleife
   //wird mindestens einmal durchlaufen
until <Bedingung>
```

Für die Bedingung gilt natürlich das gleiche wie bei der `while`-Schleife, auch hier sind also boolesche Variablen, Vergleiche oder miteinander verknüpfte Vergleiche zulässig.

Abbildung 3.3 zeigt das Ablaufschema einer `repeat`-Schleife.

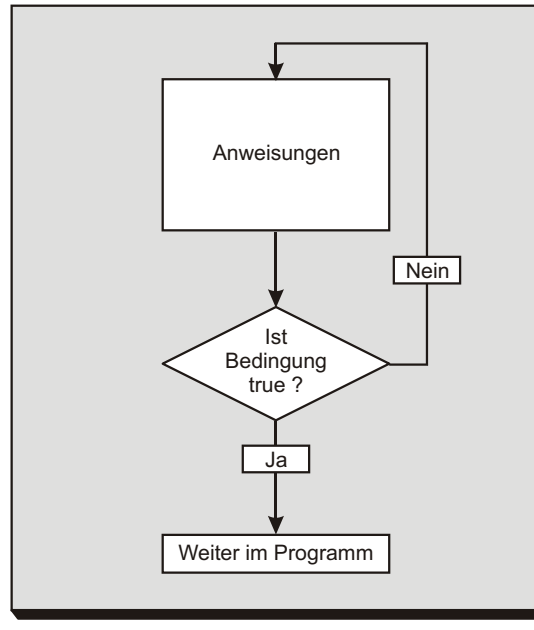


Abbildung 3.3: Das Ablaufschema der repeat-Schleife

Die Gefahr einer Endlosschleife ist natürlich bei beiden Schleifentypen gegeben. Achten Sie deshalb darauf, dass die überprüfte Bedingung keinen statischen Wert annehmen kann, da dies zu einer Endlosschleife führen würde.

Auch für die `repeat`-Schleife möchte ich Ihnen ein Beispiel geben, nämlich die Berechnung der Quersumme einer Zahl.

```
function Quersumme(aNumber: integer): integer;
begin;
  Result := 0;
  repeat
```

```
    Result := Result+ (aNumber mod 10);
    aNumber := aNumber div 10;
  until aNumber=0;
end;
```

Im Beispiel wird die Zahl innerhalb unserer `repeat`-Schleife ständig durch 10 dividiert. Dadurch dass wir die Division mit dem Operator `mod` durchführen, erhalten wir stets den Rest, wir »knabbern« also immer eine Stelle der Zahl ab und addieren sie zum Resultat. Wenn die Zahl gleich 0 ist, beenden wir die Berechnung und der Wert wird zurückgeliefert.

Abbrechen und Fortsetzen von Schleifen

Break

Die Anweisungen `break` und `continue` werden für die Abbruchsteuerung einer Schleife verwendet. Eine `for`-, `repeat`- oder `while`-Schleife kann über die Anweisung `break` verlassen werden, das Programm wird dann mit der ersten Anweisung nach der Schleifenkonstruktion fortgesetzt.

Continue

Die Anweisung `continue` *innerhalb* einer Schleife bewirkt, dass mit dem nächsten Durchlauf einer Schleife begonnen wird. Tritt innerhalb einer Schleife eine Unterbrechung durch einen ungültigen Wert auf, kann mit Hilfe der Anweisung `continue` weitergearbeitet werden, bis die Schleife komplett durchlaufen ist (es könnten sich hinter dem aktuellen Wert ja noch gültige Werte befinden).

Am sinnvollsten ist `continue` allerdings dann, wenn innerhalb der Schleife umfangreiche Berechnungen durchgeführt werden sollen. Die Annahme soll sein, dass während der Berechnung ein Zustand auftreten kann, der eine weitere Berechnung sinnlos macht. In diesem Fall soll die Schleife mit dem nächsten Durchlauf fortfahren, ohne die nachfolgenden Rechenoperationen zu berücksichtigen. In diesem Fall könnte man `continue` natürlich auch verwenden:

```
function BerechneMitSchleife: integer;
var
  Ergebnis: integer;
begin;
  for i:=1 to 100 do
  begin;

    //Rechenoperationen hierher

    if (Ergebnis-i)=0 then
      continue;
```

```
    //Weitere Rechenoperationen
  end;
end;
```

Im obigen Beispiel würde, falls während der Gesamtberechnung der Wert von `Ergebnis` gleich dem Wert von `i` ist, der nächste Durchlauf der Schleife aufgerufen.

Beide Anweisungen, `break` und `continue`, sind nur innerhalb von Schleifen gültig. Werden sie außerhalb einer Schleife aufgerufen, gibt Delphi bereits während der Kompilierung des Projekts eine Fehlermeldung aus.

Mit diesen drei Schleifenarten können Sie alle denkbaren Wiederholungen von Programmabschnitten durchführen.

3.2 Verzweigungen

Beim Beispiel mit der `while`-Schleife haben wir bereits eine Verzweigung kennen gelernt, sie allerdings noch nicht ausführlich besprochen. Delphi kennt zwei Arten von Verzweigungen, nämlich einmals abhängig von einer Bedingung und einmal abhängig vom Wert einer Variable mit ordinalem Datentyp. Letztere kann natürlich auch durch erstere imitiert werden, diese wird dann aber sehr unübersichtlich.

If-then-else

Die bekannteste Verzweigung bei fast allen gängigen Programmiersprachen ist wohl die `if-then-else`-Verzweigung. In jeder Sprache etwas anders angewendet (von der Syntax her) hat sie doch überall die gleiche Bedeutung und tritt auch fast überall am häufigsten auf.

Über das reservierte Wort `if` wird eine Bedingung ausgewertet, die entweder wahr oder falsch ist. Ist die Bedingung wahr, wird der nachfolgende Programmcode ausgeführt, ist sie falsch, wird der Programmcode nicht ausgeführt. Um auch einen bestimmten Code ausführen zu können, wenn die Bedingung falsch ist, wird das reservierte Wort `else` verwendet. Die Syntax der `if-then-else`-Verzweigung sieht folgendermaßen aus:

Syntax

```
if <Bedingung> then
begin;
  //Anweisungen, wenn Bedingung wahr
end
else
begin;
  //Anweisungen, wenn Bedingung falsch
end;
```

Zu beachten ist, dass vor dem reservierten Wort `else` kein Semikolon stehen darf. Das Ablaufschema für eine `if-then-else`-Verzweigung sehen Sie in Abbildung 3.4.

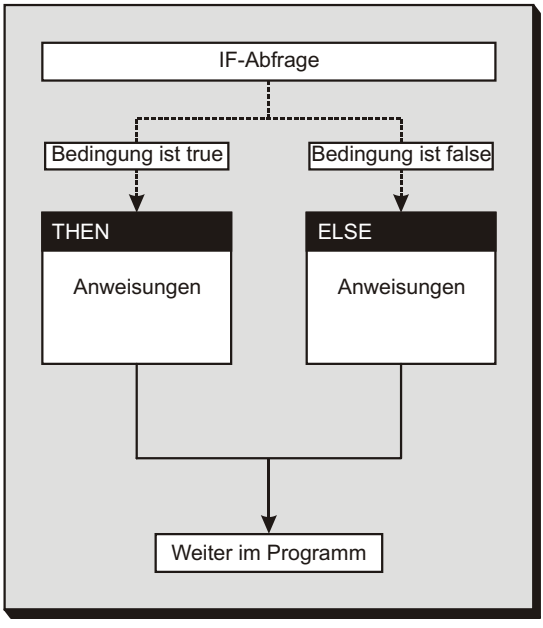


Abbildung 3.4: Das Ablaufschema zur if-then-else-Verzweigung

Natürlich können Sie auch hier wieder die logischen Operatoren benutzen, um mehrere Bedingungen miteinander zu verknüpfen.

Case

Eine weitere Möglichkeit der Verzweigung bzw. des Ausführens von Programmcode abhängig von einer Bedingung ist die `case`-Anweisung. Die `case`-Anweisung wertet keine wahr/falsch-Bedingung aus, sondern einen ordinalen (Aufzählungs-)Typ und verzweigt entsprechend dessen Position bzw. Wert. Die Syntax der `case`-Anweisung liest sich wie folgt:

Syntax

```
case (Aufzählungstyp) of
  (1. Wert): begin;
               //Code für 1. Wert des ordinalen Typs
             end;
  (2. Wert): begin;
               //Code für 2. Wert des ordinalen Typs
             end;
else
  //Anweisungen, wenn kein Wert übereinstimmt
end; {Case}
```

Die case-Anweisung verzweigt also entsprechend dem Wert der Zählvariablen. Wenn kein Wert übereinstimmt, wird in den else-Teil verzweigt, sofern dieser vorhanden ist; andernfalls wird nach der case-Anweisung weitergemacht.

Im Falle der case-Anweisung steht vor dem else ein Komma, die Syntax ist durchaus korrekt. Das else gehört nämlich nicht zu der Anweisung innerhalb von case, sondern zur case-Anweisung selbst. Wenn else im Zusammenhang mit der if-Anweisung verwendet wird, darf das Komma nicht stehen.

Das Ablaufschema einer case-Anweisung sehen Sie in Abbildung 3.5.

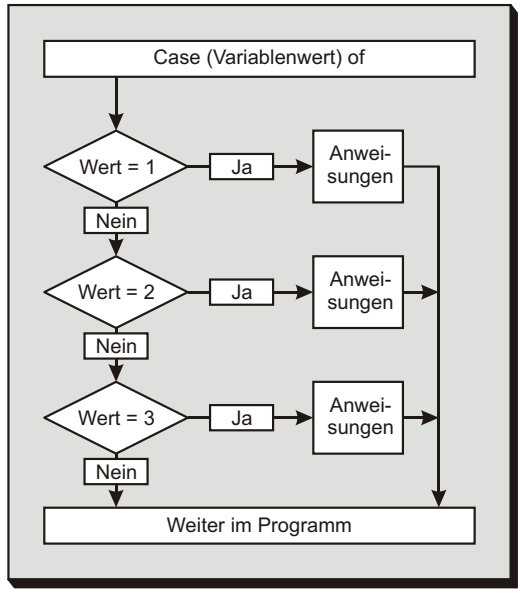


Abbildung 3.5: Ablaufschema einer case-Anweisung

Ein Beispiel für eine case-Auswertung wäre die Ausgabe der Anzahl der Tage eines Monats. Programmiert mithilfe von case sähe eine solche Funktion folgendermaßen aus:

```
function GetMonthDays(Month: byte): byte;
begin;
  Case Month of
    2: Result := 28;
    4,
    6,
    9,
    11: Result := 30;
  else
    Result := 31;
```

Um zu betonen, dass diese Auswertung auch mit einer `if`-Anweisung möglich ist, habe ich diese im nächsten Beispiel verwendet. Sie werden allerdings sofort sehen, dass sie viel schlechter zu lesen ist:

```
function GetMonthDays(Month: byte): byte;
begin;
  Result := 31;
  if Month=2 then
    Result := 28
  else
  if Month in [4,6,9,11] then
    Result := 30;
end;
```

3.3 Übungen

Übung 1

Erstellen Sie eine Schleife, die drei mal von 0 bis 10 und wieder zurück zählt.

Übung 2

Erstellen Sie eine Funktion `kgv`, die das kleinste gemeinsame Vielfache zweier übergebener Zahlen zurückliefert.

Übung 3

Schreiben Sie eine Funktion `GetFaculty`, die die Fakultät einer Zahl zurückliefert. Die Fakultät einer Zahl lässt sich sehr leicht berechnen, indem alle Zahlen von 1 bis zum Zahlenwert miteinander multipliziert werden. Damit ergibt sich z.B. für die Zahl 5 eine Fakultät von 125, denn:

$$F_5 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 125$$

Übung 4

Schreiben Sie eine Funktion `CheckPrime`, mit der Sie kontrollieren können, ob eine Zahl eine Primzahl ist.

Übung 5

Schreiben Sie ein kleines Programm mit einer Listbox, in der Sie alle Primzahlen bis zu einem vorgegebenen Wert ausgeben können.

Übung 6

Schreiben Sie eine Funktion `GetBiggerNr`, der Sie zwei ganze Zahlen als Parameter übergeben. Die größere der beiden Zahlen soll zurückgeliefert werden. Betten Sie die Funktion in ein Programm ein, mit dem Sie einen Test durchführen können.

Übung 7

Ein wenig Mathematik: Wie wir alle wissen, ist die Zahl π eine Zahl mit unendlich vielen Nachkommastellen. Damit war (und ist) sie auch sehr beliebt bei Mathematikern. Bis heute wird an dieser Zahl gerechnet, um möglichst viele Nachkommastellen herauszufinden.

Der Mathematiker Leibniz hat die folgende Methode zur Berechnung von π gefunden:

$$\pi = \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots + \frac{1}{x}\right) \cdot 4$$

Setzen Sie die Methode von Leibniz in einem Programm um. Das maximale *x* soll angegeben werden können (Sie können es ruhig sehr hoch wählen, wenn der Algorithmus richtig programmiert wurde). Will man π genau berechnen, müsste *x* übrigens am Ende unendlich sein.

Übung 8

Auch diese Übung ist ein wenig umfangreicher. Es gilt auch hier, ein komplettes Programm zu erstellen.

Zu berechnen ist die Quadratwurzel einer Zahl nach dem Annäherungsprinzip. Das Prinzip ist sehr alt, aber auch sehr leicht umzusetzen. Es geht um eine Quadratwurzel-Berechnung nach Heron.

Heron ging von folgender Überlegung aus: Wenn ein Quadrat existiert, dessen Fläche meiner ursprünglichen Zahl entspricht, so muss die Seitenlänge des Quadrats auch der Wurzel meiner Zahl entsprechen. So hat z. B. ein Quadrat mit der Seitenlänge 2 eine Fläche von 4, was exakt Herons Gedankengang entspricht.

Um nun die Quadratwurzel jeder beliebigen Zahl zu berechnen, bildet man zunächst ein Rechteck, wobei eine Seite des Rechtecks die Länge »1« aufweist, die Länge der zweiten Seite ist so groß wie die Zahl, deren Quadratwurzel gesucht wird. Nun versucht man, durch Annäherung aus

dem Rechteck ein Quadrat zu machen. Dazu bildet man zunächst das arithmetische Mittel aus den beiden Seiten. Da der Flächeninhalt gleich bleibt und eine Seite ja neu berechnet ist, kann man nun auch die andere Seite neu berechnen, denn das Produkt aus beiden Seiten muss ja den Flächeninhalt ergeben. Man erhält also zwei neue Werte.

Danach beginnt man von vorn, errechnet wieder das arithmetische Mittel beider Seiten und erhält wiederum eine genauere Seitenlänge. Irgendwann ist das Ergebnis dann genau genug, d.h. die Seitenlängen sind annähernd gleich und entsprechen damit der Quadratwurzel der Fläche.

Die Berechnung des arithmetischen Mittels *a'* aus zwei Seiten *a* und *b* stellt sich wie folgt dar:

$$a' = \frac{(a+b)}{2}$$

a' entspricht der neuen Seitenlänge. Die neue Seitenlänge der zweiten Seite des Rechtecks wird dementsprechend über die Fläche des Rechtecks berechnet, die hier mit groß *A* angegeben ist:

$$b' = \frac{A}{a'}$$

Danach entsprechen *a'* und *b'* den neuen Seitenlängen. *a'* ist der Annäherungswert.

Mit diesen Informationen sollten Sie in der Lage sein, ein Programm zu schreiben, mit dessen Hilfe die Quadratwurzel nach dem Heron'schen Prinzip berechnet werden kann. Lagern Sie die eigentliche Berechnung in eine Prozedur aus, damit sie auch in anderen Applikationen benutzt werden kann.

Die eigentliche Berechnung ist trivial. Daher noch einige Vorgaben für diese Aufgabe.

- Der Anwender soll eine Fehlertoleranz angeben können, also einen Wert der benötigten Genauigkeit. Ist das Ergebnis der Berechnung genau genug, wird nicht weitergerechnet.
- Wird keine Fehlertoleranz angegeben, soll das Programm automatisch mit einer Fehlertoleranz von 0,00004 rechnen.
- Die Vorgaben sind relativ leicht zu erfüllen. Achten Sie auch darauf, was passiert, wenn der Anwender eine negative Zahl eingibt. Immerhin wollen wir eine Wurzel berechnen.

Übung 9

In dieser Übung soll es um eine andere Art der Berechnung des ggT gehen, nämlich um die Berechnung mit Hilfe des euklidischen Algorithmus. Dieses Verfahren wurde schon vor sehr langer Zeit entdeckt, ist aber sehr effektiv.

Bei zwei Zahlen x und y beruht der Algorithmus auf der Tatsache, dass im Fall *x>y* der ggT von *x* und *y* gleich dem ggT von *y* und *x-y* ist. Das ist nun mal so. Ihre Aufgabe soll nun darin bestehen, diesen Algorithmus in eine Funktion umzusetzen, die den ggT zweier übergebener Zahlen mit Hilfe des euklidischen Algorithmus berechnet.

Die Schwierigkeit bei dieser Übung besteht eigentlich nicht im Algorithmus selbst, sondern im Finden des richtigen Ansatzpunkts. Der euklidische Algorithmus ist eigentlich ein Kontroll- und Vertauschalgorithmus. Wenn *y* größer ist als *x*, vertauschen wir die beiden. Dann berechnen wir *x* neu mithilfe der Formel *x-y*. Sie müssen sich nur überlegen, wann eine Lösung gefunden ist. Zugegebenermaßen etwas schwierig.

3.4 Tipps

Tipps zu Übung 1

- Verwenden Sie nicht die `for`-Schleife.

Tipps zu Übung 2

- Ein kleinstes gemeinsames Vielfaches existiert immer.
- Das kgV ist der Wert, der, wenn er durch einen der beiden gegebenen Werte dividiert wird, den Rest null ergibt.
- Die erste Möglichkeit für das kgV ist der größere der beiden Werte. Damit haben Sie einen Anfangswert.

Tipps zu Übung 4

- Eine Primzahl ist nur durch sich selbst und durch 1 teilbar. Die Zahl 2 ist allerdings keine Primzahl, obwohl sie diese Bedingungen erfüllt.
- Sie müssen lediglich eine Schleife bauen, in der alle Zahlen bis zum zu kontrollierenden Wert durchlaufen werden. Durch die Schleifenvariable wird versucht zu dividieren. Ist der Rest einer dieser Divisionen gleich 0, so handelt es sich nicht um eine Primzahl.

Tipps zu Übung 7

- Analysieren Sie die Gleichung. Möglicherweise finden Sie etwas, was man in eine Schleife packen könnte ...
- Zum Umkehren eines Vorzeichens können Sie die Multiplikation mit -1 benutzen.
- Benutzen Sie eine Variable für den Nenner, die Sie mit jedem Durchlauf um 2 erhöhen. Dann müssen Sie auch noch mit jedem Durchlauf das Vorzeichen wechseln.

Tipps zu Übung 8

- Die Benutzung der angegebenen Formeln sollte kein Problem sein. Überlegen Sie, mit welcher Schleifenart Sie den Fehlerwert in die Kontrolle einbringen können.
- Das Ergebnis ist erreicht, wenn der Unterschied zwischen den beiden errechneten Seiten kleiner oder gleich des Fehlerwertes ist.
- Verwenden Sie entweder eine `while`- oder eine `repeat`-Schleife.

Tipps zu Übung 9

- Mit einer `repeat`-Schleife lässt sich der Algorithmus leicht implementieren.
- Ziehen Sie so lange den kleineren vom größeren Wert ab, bis der Wert 0 erreicht ist. Der andere Wert entspricht dann dem Wert des ggT.
- Bilden Sie eine `repeat`-Schleife, die zunächst kontrolliert, ob Wert 1 größer als Wert 2 ist. Wenn nicht, werden die Werte vertauscht. Dann wird Wert 2 von Wert 1 abgezogen. Die `repeat`-Schleife beginnt von vorn. Die Schleife wird beendet, wenn Wert 1 kleiner oder gleich 0 ist. Ist Wert 1 kleiner 0, gibt es keinen ggT.

3.5 Lösungen

Lösung zu Übung 1

Wir wollen in dieser Übung eine Schleife erstellen, die drei Mal von 0 nach 10 und zurück zählt. Normalerweise denkt man hierbei immer an eine `for`-Schleife. Das wäre allerdings eine ungünstige Wahl, da wir bei dieser Schleifenart zwar beide Zählrichtungen verwenden, aber nicht zwischen den beiden umschalten können. Verwenden wir also entweder eine `while`-Schleife oder eine `repeat`-Schleife.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  theValue : integer;
  count    : integer;
  direction: integer;
begin
  direction := 1;
  count := 3;
  theValue := 0;
  repeat
    theValue := theValue+direction;
    if theValue=0 then
      dec(count);
    if (theValue=0) or (theValue=10) then
      direction := direction*-1;
  until Count=0;
end;
```

Die Variable `direction` gibt die Zählrichtung an. Um die Richtung umzukehren, müssen wir `direction` lediglich mit -1 multiplizieren. Mithilfe der Variable `count` zählen wir, wie oft wir die Schleife haben laufen lassen. Wir müssen dazu lediglich `count` herabzählen, bis wir bei 0 angekommen sind, dann hören wir auf. `theValue` enthält unseren aktuellen Zählwert.

Das Ganze funktioniert natürlich auch mit einer `while`-Schleife. Es ändert sich ja nur die Position der Abfrage, die bei der `while`-Schleife am Anfang vorgenommen wird.

Lösung zu Übung 2

Den größten gemeinsamen Teiler haben wir bereits berechnet, als Beispiel in diesem Kapitel. Berechnen wir nun noch das kleinste gemeinsame Vielfache. Dabei wissen wir, dass es immer ein kleinstes gemeinsames Vielfaches gibt – nämlich bei zwei Werten a und b im ungünstigsten Fall das Produkt der beiden. Damit entfällt die Möglichkeit, kein Ergebnis zu bekommen.

```
function kgv(v1, v2: integer): integer;
var
  helpVal : integer;
  kgvOK   : Boolean;
begin;
  helpVal := v1;
  repeat
```

```
    kgvOK := (((helpVal mod v1)=0) and
             ((helpVal mod v2)=0));
    if kgvOK then
      Result := helpVal
    else
      inc(helpVal);
  until kgvOK;
end;
```

Die Variable `helpVal` dient zur Kontrolle des kgV, eigentlich enthält sie immer den Wert, der das kgv darstellen soll und überprüft wird. Ist die Überprüfung erfolgreich, wird der Wert zurückgeliefert und die Schleife beendet.

Abbildung 3.6 zeigt einen Screenshot des Programms. Sie finden es auf der beiliegenden CD im Verzeichnis *Übungen\Kapitel03\Übung_2*.

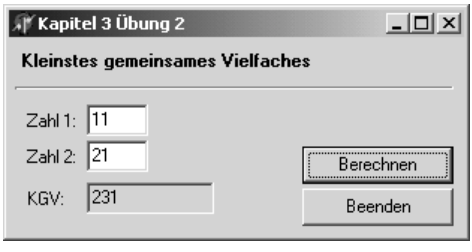


Abbildung 3.6: Berechnung des kgV im Programm

Lösung zu Übung 3

Die Fakultät eines Werts zu berechnen ist nicht weiter schwer. Alles, was wir dazu benötigen, ist eine simple `for`-Schleife, die bis zum gegebenen Wert hochgezählt wird. Innerhalb der Schleife wird dann der jeweilige Wert der Laufvariablen mit dem aktuellen Ergebniswert multipliziert.

Für diese Übung habe ich statt des Datentyps `integer` den Datentyp `Cardinal` verwendet. Der Grund ist, dass ich damit einen höheren Wert bei der Berechnung erhalte, denn `integer` ist ein vorzeichenbehafteter Typ – entweder 32 Bit oder als `Int64` mit 64 Bit Breite – und Cardinal ein 32-Bit-vorzeichenloser Typ. Unsere Fakultät ist ohnehin immer positiv, also benötigen wir kein Vorzeichen.

```
function Faculty(v: Cardinal): Cardinal;
var
  i: Cardinal;
begin;
  Result := 1;
```

```
  for i:=1 to v do
    Result := Result*i;
end;
```

Die Funktion sollte kein Problem darstellen. Sie finden das Programm mit der Lösung auf der beiliegenden CD im Verzeichnis *Übungen\Kapitel03\Übung_3*.

Lösung zu Übung 4

Zunächst zu den Grundlagen, was Primzahlen betrifft. Eine Zahl ist dann eine Primzahl, wenn sie nur durch 1 und sich selbst teilbar ist. Da eine Zahl immer durch 1 und immer durch sich selbst teilbar ist (was 1 ergibt), können wir diese beiden Punkte von der Berechnung ausnehmen.

```
function CheckPrime(v: integer): Boolean;
var
   I: integer;
begin;
   Result := false;
   if (v<3) then
      exit;
   for i:=2 to v-1 do
      if (v mod i)=0 then
         Result := true;
   Result := not Result;
end;
```

In der Funktion wird kontrolliert, ob die Zahl durch eine zwischen 1 und dem Zahlenwert liegende Zahl teilbar ist. Ist dem so, wird `Result` auf `true` gesetzt, es handelt sich *nicht* um eine Primzahl. Falls dem nicht so ist, bleibt `Result` auf dem vorgegebenen Wert, den wir mit `false` initialisiert haben.

Der Rückgabewert ist nun `true`, wenn die übergebene Zahl keine Primzahl ist, und `false`, wenn es sich um eine Primzahl handelt. Die Funktion soll aber genau andersherum arbeiten, also `true` zurückgeben, wenn der übergebene Wert eine Primzahl ist. Daher müssen wir am Ende der Funktion den Wert von `Result` noch negieren, was über den Operator `not` geschieht.

Am Anfang der Funktion werden die Werte 1 und 2 gleich abgeblockt, denn diese können ohnehin nie eine Primzahl sein. Die Funktion würde aber zumindest bei dem Wert 2 ergeben, dass es sich um eine Primzahl handelt. Wir müssen diese Werte also von der Berechnung ausschließen. Aus diesem Grund muss die Funktion auch »falsch herum« arbeiten.

Auch zu dieser Übung ist die Anwendungsprogrammierung trivial, daher auch hier nicht weiter aufgeführt. Den Quelltext zu dieser Übung finden Sie auf der CD im Verzeichnis *Übungen\Kapitel03\Übung_4.*

Lösung zu Übung 5

Das Hauptelement der Übung – die Kontrolle, ob eine Zahl eine Primzahl ist – übernehmen wir natürlich aus Übung 4. Damit bleibt uns nur noch, eine Routine zu schreiben, die mittels einer Schleife alle Zahlen innerhalb des vorgegebenen Rahmens überprüft.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NoAction: Boolean;
  Value1,
  Value2,
  i       : Integer;
begin
  NoAction := (Edit1.Text='') OR (Edit2.Text='');
  if NoAction then
     exit;
  try
    Value1 := StrToInt(Edit1.Text);
    Value2 := StrToInt(Edit2.Text);
  except
    ShowMessage('Ein Fehler ist aufgetreten');
  end;
  if Value1>Value2 then
    Swap(Value1,Value2);
  //ListBox leeren
  ListBox1.Items.Clear;
  //Jetzt alle Zahlen kontrollieren
  for i:=Value1 to Value2 do
    if IsPrimZahl(i) then
      ListBox1.Items.Add(IntToStr(i));
end;
```

Zunächst werden die Werte aus den Eingabefeldern übernommen, in diesem Fall auch mal begleitet von einem Schutzblock, damit wir Falscheingaben abfangen können. Für unsere Schleife, die wir für die Berechnung verwenden wollen, müssen wir aber sicherstellen, dass `Value1` den niedrigeren Wert enthält. Also kontrollieren wir dies und drehen die Werte gegebenenfalls um. Das erledigt die Prozedur `Swap`.

```
procedure Swap(var a: Integer; var b: Integer);
var
  c: Integer;
begin;
  c := a;
  a := b;
  b := c;
end;
```

Die eigentliche Auswertung geschieht in den letzten fünf Zeilen der Routine. Zunächst wird der Inhalt der Listbox geleert. Dann benutzen wir eine `for`-Schleife, um alle Zahlen vom Startwert bis zum Endwert darauf zu kontrollieren, ob es sich um Primzahlen handelt. Wenn ja, wird die Zahl zur Listbox hinzugefügt, andernfalls nicht. Achten Sie darauf, dass die Zahl in einen String umgewandelt werden muss, bevor sie der Listbox hinzugefügt werden kann.

Den Quelltext zu dieser Übung finden Sie auf der CD im Verzeichnis *Übungen\Kapitel_3\Übung_5.* Abbildung 3.7 zeigt einen Screenshot des laufenden Programms.

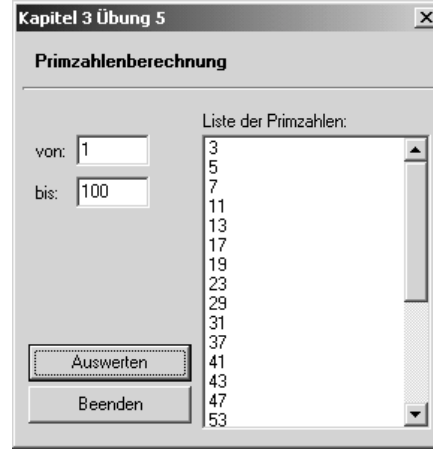


Abbildung 3.7: Primzahlen-berechnung

Es bleibt anzumerken, dass diese Methode, mehrere Primzahlen zu ermitteln, nicht besonders effektiv ist. Eine weitaus effektivere Möglichkeit, genannt das *Sieb des Erasthotenes*, werden wir im Kapitel über Algorithmen kennen lernen.

Lösung zu Übung 6

Die Lösung zu dieser Übung erfordert eigentlich keine große Mühe.

```
function IsBigger(v1,v2: integer): integer;
begin;
  if v1>v2 then
    Result := v1
  else
    Result := v2;
end;
```

Eine simple `if`-Anweisung bringt uns das gewünschte Resultat.

Lösung zu Übung 7

In dieser Übung gilt es, eine mathematische Formel in ein Computerprogramm umzusetzen, wenn Sie so wollen, einen Algorithmus für ein Problem zu finden. Machen wir uns ein paar Gedanken zu den gegebenen Werten.

Zunächst wissen wir, dass sich das Vorzeichen innerhalb der Klammer ständig ändert. Wir wissen bereits, wie wir eine solche Vorzeichenänderung bewerkstelligen können, nämlich durch die Multiplikation mit dem Wert -1. Wir deklarieren uns also eine Variable, in diesem Fall `sign`, die das aktuelle Vorzeichen darstellt.

Als Nächstes sehen wir uns den Nenner an. Auch für diesen deklarieren wir uns eine Variable. Da es sich um den Wert handelt, der ständig erhöht wird, der also die Iterationen darstellt, nennen wir die Variable `iter`.

Wir beginnen innerhalb der Klammer mit dem Nenner 3, was also den Anfangswert für die Variable `iter` darstellt. Dann müssen wir nur noch darauf achten, dass `iter` immer um den Wert 2 erhöht wird und dass nach jeder Erhöhung des Werts das Vorzeichen gewechselt wird.

Damit wir auch einen möglichst genauen Wert erhalten, bauen wir noch eine Genauigkeit ein, sozusagen die maximale Anzahl der Iterationen. Je höher dieser Wert ist, desto näher kommen wir an den wirklichen Wert von π. Ein sinnvoller Wert liegt bei einer Genauigkeit von etwa 10.000.

Der Quellcode sollte relativ leicht verständlich sein:

```
function pi_Leibniz(n: integer): double;
var
  sign : integer;
  iter : integer;
```

```
begin;
  vorz   := -1;
  Result := 1;
  iter := 3;
  while iter<n do
  begin;
    Result := Result + (vorz*(1/iter));
    vorz := vorz*(-1);
    inc(iter,2);
  end;
  Result := Result*4;
end;
```

Das Programm finden Sie natürlich ebenfalls auf der beiliegenden CD, im Verzeichnis *Übungen\Kapitel03\Übung_7*. Einen Screenshot sehen Sie in Abbildung 3.8.

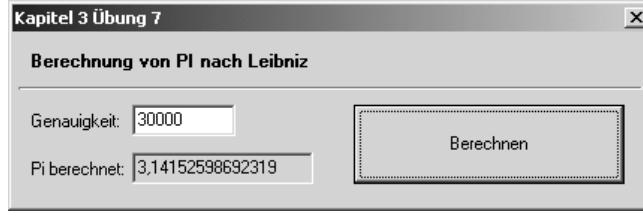


Abbildung 3.8: Berechnung von Pi nach Leibniz

Lösung zu Übung 8

Die Lösung zu dieser Aufgabe ist leichter, als es zunächst den Anschein hat. Wir werden folgendermaßen vorgehen.

Die Fehlertoleranz wird als typisierte Konstante implementiert und gleich initialisiert. Das hat den Vorteil, dass der Toleranzwert gleich bei der Deklaration initialisiert werden kann. Weiterhin kann der Anwender durch seine Eingabe den Wert ändern, es handelt sich ja um eine *typisierte* Konstante. Natürlich könnte der Toleranzwert ebenso gut über eine Variable implementiert werden.

Nun wird die Zahl, deren Quadratwurzel berechnet werden soll, aus dem Eingabefeld entnommen. Da eine Zahl, aus der die Quadratwurzel gezogen werden muss, immer positiv sein muss, müssen wir diese erst kontrollieren und gegebenenfalls eine positive Zahl daraus machen. Wir kehren die negative Zahl einfach um, indem wir sie mit dem Wert -1 multiplizieren.

Nun wird noch die Fehlertoleranz übernommen, falls der Anwender eine eingegeben hat. Dann folgt die Berechnung. Hier die Prozedur im Zusammenhang:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Zahl   : double;
   x,y    : double;
const
   Fehler : double = 0.000004;
begin
   ListBox1.Items.Clear;
   //Werte aus Edit-Feldern
   try
      Zahl := StrToFloat(Edit1.text);
      if Zahl<0 then
      begin;
         Zahl := Zahl*(-1);
         Edit1.Text := FloatToStr(Zahl);
      end;
      if Edit2.text<>'' then
         Fehler := StrToFloat(Edit2.text);
   except
     ShowMessage('Ein Fehler ist aufgetreten');
   end;

  //Quadratwurzel nach Heron
   x := 1;
   y := Zahl;
   repeat
      y := (x+y)/2;
      x := Zahl/y;
      ListBox1.Items.Add(FloatToStr(x));
   until (y-x)<Fehler
end;
```

Die Berechnung wird entsprechend der Angaben in der Übungsbeschreibung vorgenommen. Die `repeat`-Schleife wiederholt die Berechnung so lange, bis der Unterschied zwischen den Seitenlängen des Quadrats nicht mehr größer ist als der Fehlertoleranzwert. Das Zwischenergebnis jeder Berechnung wird dann der Listbox hinzugefügt.

Sie werden beim Testen der Funktion feststellen, dass es sich um eine sehr schnelle und sehr genaue Berechnung handelt. Die Genauigkeit

richtet sich allerdings auch nach der Art des Datentyps, den wir wählen. Ich habe für dieses Beispiel aus gutem Grund den Typ `double` gewählt.

Den Quelltext zu dieser Übung finden Sie auf der CD im Verzeichnis *Übungen\Kapitel03\Übung_8.*

Die Berechnung einer Wurzel nach der Heron'schen Methode ist recht effektiv, mit wenigen Iterationen wird bereits ein recht genauer Wert ermittelt. Einen Screenshot des fertigen Programms sehen Sie in Abbildung 3.9.

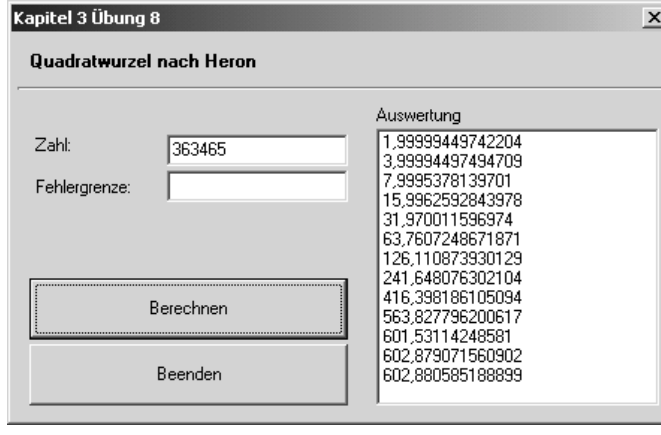


Abbildung 3.9: Quadratwurzelberechnung nach Heron

Lösung zu Übung 9

Die Aufgabe war zwar als schwer tituliert, aber es war eigentlich schwieriger, die richtige Umsetzung zu finden, als die Funktion zu programmieren. Hier ist der euklidische Algorithmus für die Bestimmung des ggT:

```
function GetGGT(Value1, Value2: integer): integer;
begin
  Result := -1;
  repeat
    if Value1<Value2 then
      Swap(Value1,Value2);
    Value1 := Value1 - Value2;
  until Value1<=0;
  if Value1=0 then
    Result := Value2;
end;
```

Die Prozedur `Swap` vertauscht lediglich die beiden Variablen, die `if`-Schleife am Ende kontrolliert nur, ob es ein Ergebnis gab oder nicht. Den kompletten Quelltext zu dieser Übung finden Sie auf der CD im Verzeichnis *Übungen\Kapitel03\Übung_9*. Abbildung 3.10 zeigt das Programm zur Laufzeit.

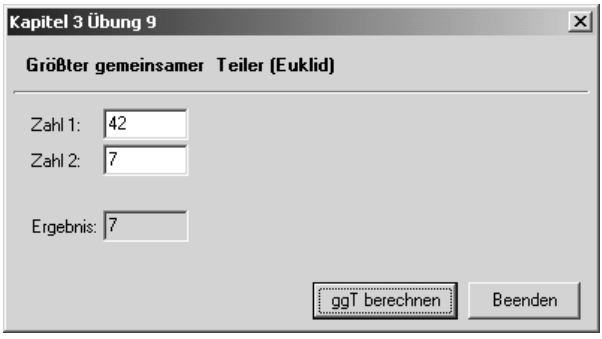


Abbildung 3.10: Berechnung des größten gemeinsamen Teilers nach Euklid